

Mallard BASIC

For the Amstrad PCW8256/8512 & PCW9512



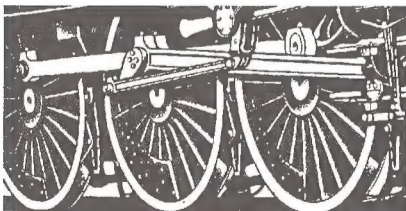
**LOCOMOTIVE
SOFTWARE**

Introduction
and Reference

Mallard BASIC

Introduction and Reference

The world speed record for a steam locomotive is held by LNER 4-6-2 No. 4468 "Mallard", which hauled seven coaches weighing 240 tons over a measured quarter mile at 126 mph (202 kph) on 3rd July 1938.



**LOCOMOTIVE
SOFTWARE**

© Copyright 1987 Locomotive Software Limited
All rights reserved.

Neither the whole, nor any part of the information contained in this manual may be adapted or reproduced in any material form except with the prior written approval of Locomotive Software Limited.

While every effort has been made to verify that this software works as described, it is not possible to test any program of this complexity under all possible circumstances. Therefore Mallard BASIC is provided 'as is' without warranty of any kind either express or implied.

The particulars supplied in this manual are given by Locomotive Software in good faith. However, Mallard BASIC is subject to continuous development and improvement, and it is acknowledged that there may be errors or omissions in this manual.

Locomotive Software reserves the right to revise this manual without notice.

Written by Locomotive Software Ltd and Ed Phipps Documentation Services
Produced and typeset electronically by Locomotive Software Ltd
Printed by Grosvenor Press (Portsmouth) Ltd

Published by Locomotive Software Ltd
Allen Court
Dorking
Surrey RH4 1YL

2nd Edition Published 1987 (Reprinted with corrections May 1989)
ISBN 185195 009 5

Mallard BASIC is a trademark of Locomotive Software Ltd
LOCOMOTIVE is a registered trademark of Locomotive Software Ltd
AMSTRAD is a registered trademark of AMSTRAD plc
IBM is a registered trademark of International Business Machines Corp
CP/M-80, CCP/M-86 and MP/M-86 are trademarks of Digital Research Inc
MS-DOS is a trademark of Microsoft® Corporation
VT52 is a trademark of Digital Equipment Corp

Preface

This book describes how to use Locomotive Software's Mallard BASIC interpreter to write and use BASIC programs on your Amstrad PCW.

BASIC was originally designed many years ago as a programming language that was simple to learn, as an aid to teaching computer programming. There are now many versions of BASIC, developed from this original.

Mallard BASIC is a powerful modern BASIC with a number of significant improvements. In particular, Mallard BASIC has a very unusual facility (for BASICs on microcomputers) for indexing data stored in a file, and automatically keeping this index in alphabetical order. This is a very useful facility to have in a program that works with a database.

BASIC is very versatile and can be used to perform most tasks involving manipulating numbers and text. The simplest use of BASIC is as a sophisticated calculator, but you can quickly move on from this to using BASIC to write programs. Programs can perform an infinite range of tasks, from simple things such as totalling sequences of figures to complex tasks such as maintaining a personnel file, calculating payrolls or performing statistical analyses.

You can use programs written by others (of which a wide range are available) or write your own, once you have learned how to program.

The copy of BASIC on your CP/M disc is set up ready for use on your machine. To use it, you just have to load CP/M and type the command BASIC `RETURN`. You then have access to all the facilities of BASIC described in this manual – including the screen-based editor which displays your program lines as you change them.

About this manual

This manual is designed to help you use Mallard BASIC. It is divided into two parts:

Part I: A gentle introduction, suitable for new users including those who have never written a computer program before.

Part II: A reference section containing a detailed description of all the facilities of BASIC.

The Introduction takes the user from using BASIC as a simple calculator, through writing simple programs (for example, to print the result of a simple calculation), to preparing fully-featured programs that manipulate complex data files. The Reference section describes BASIC in detail. It is aimed at experienced users who have used other BASICs before or who have read and understood the Introduction.

Many of the operations described in this manual are concerned with manipulating numbers and text, but not all. For example, we show you how to position the cursor on the screen or clear the screen and how to control the printer from your program – setting the page length, positions of tabs and even resetting the printer from inside your BASIC program.

We also show you how to incorporate machine language routines in your program and how to call up external routines, such as those within the operating system itself. For example, we show you how to use these techniques to generate graphics effects by using GSX, the graphics extension to CP/M.

If you are entirely new to BASIC

Start by reading through the Introduction, working through the examples on your computer as you do so. You may, if you wish, just read up to the end of Chapter 5 if you don't want to work with files initially. By the end of the Introduction, you should be able to write simple but useful BASIC programs with occasional help from the Reference section.

To discover how to use the full range of facilities offered by Mallard BASIC, you should then read Part II. Although Part I covers the most useful facilities of BASIC, it does so at the fairly elementary level, with the emphasis on understanding rather than complete technical detail. Part II gives the definitive description of Mallard BASIC.

If after reading Part I you need more help to understand programming, you should turn to one of the many excellent tutorial texts that are available. Ask your dealer for details.

If you have programmed in BASIC before

You probably won't need to read Part I in any detail, except perhaps for Chapter 7 which describes the very powerful Keyed File Handling (JETSAM) provided by Mallard BASIC. Concentrate instead on the Reference section which is intended to describe Mallard BASIC, fully and accurately. This includes in Chapter 10 the formal specification of each of the commands and functions in keyword order.

Different versions of Mallard BASIC

Mallard BASIC is available in different versions for use with different operating systems working on different processors. In particular, there are separate versions for 8-bit processors like the Z80 and 8080 (Mallard-80), for 16-bit processors like the 8088 and 8086 (Mallard-86); and there are multi-user versions of Mallard-86 for use on networks of 16-bit machines. There are also Run-Only versions available.

Part I of this manual is tailored to the version of Mallard BASIC supplied on the Amstrad PCW machines, but Part II can be used with any version of Mallard BASIC. However, you should note that a few of the commands only have an effect in certain versions. Where this is the case, this is mentioned in the text.

At the time of printing, the following versions are available:

- 8-bit Single User version (Mallard-80)

 - for CP/M 2.2 and CP/M Plus

- 16-bit Single User versions (Mallard-86)

 - for MS-DOS version 1

 - for MS-DOS version 2 and above

 - for CP/M-86

- 16-bit Multi-User versions (Multi-User Mallard-86)

 - for MS-DOS 3.2 and above with MS-NET

 - for TurboDOS

 - for Concurrent CP/M and Concurrent DOS

Note that Multi-User versions are only available for 16-bit machines.

Conventions

Throughout this description of BASIC, various conventions are used to represent different types of information. The principal conventions are as follows:

INPUT, PRINT, etc.

The names of BASIC's commands and functions (INPUT, PRINT etc.) are shown in this manual in capital letters. They are also always displayed in capitals when programs are listed. It does not matter whether you type these names as capitals or as lower case letters: they are automatically converted to capitals when BASIC stores the program.

see GOTO, see FIND\$, etc.

Further information is given where the given topic (GOTO, FIND\$ etc.) is described. Use the index to find the relevant page to look at.

text in this style

This style of text is used to indicate something to type or something displayed on the screen, in order to distinguish it from descriptive text.

descriptions-in-italic

Italic text is used for descriptions of the type of information that is required, rather than the information itself. For example, *number* could be 1 or 2 or 3 etc. If the description of a single item is more than one word long, the words are joined by hyphens: for example, *line-number*.

The majority of descriptions are explained in the accompanying text. The remainder are self-explanatory. The common descriptions are listed in the introduction to Chapter 10 in Part II, the overview of BASIC's commands and functions.

[items in slanted brackets]

Slanting brackets round an item are used to show that this item is optional. Normally, including the item and omitting the item produce different effects. The brackets themselves should always be omitted.

RETURN, etc.

These graphics are used to represent individual keys on the keyboard. For example, RETURN represents the Return key – not the letters R E T U R N.

(Additional conventions used in the Reference Section are described at the beginning of Part II.)

Part I

INTRODUCTION

CONTENTS

1.	Starting with BASIC	1
1.1	Entering BASIC	1
1.2	Leaving BASIC	1
1.3	Using BASIC	2
1.4	Giving BASIC commands	2
1.5	Types of information	3
1.6	Chapter review	7
2.	Starting programming with BASIC	9
2.1	A first program	9
2.2	More complex programs	11
2.3	Changing a program	12
2.4	Chapter review	14
3	Real programming	15
3.1	Designing programs	15
3.2	Getting information	16
3.3	How information is stored	20
3.4	Outputting the results	27
4	Building larger programs	37
4.1	Compartmenting a program	37
4.2	Sequences and loops	38
4.3	Making decisions	40
4.4	Stopping the program	44
4.5	Organising the program	45

5.	Manipulating information	53
5.1	Manipulating numeric information	53
5.2	Manipulating textual information	56
5.3	Converting between different types of information	59
6	Using discs for information storage	61
6.1	General disc commands	62
6.2	Sequential access files	63
6.3	Random access files	76
7	Keyed access files for data bases	89
7.1	Writing a program that uses Keyed files	91
7.2	Working out the main program	92
7.3	Preparatory stage	93
7.4	Adding a record	95
7.5	Reading a record	99
7.6	Deleting an entry	101
7.7	Closing the Keyed file	103
7.8	Enhancements	105
8.	Machine level operations	115
8.1	General information	115
8.2	Applying a patch	116
8.3	Loading a machine code program	117
8.4	Using a machine code program	119

Starting with BASIC

This chapter tells you how to load BASIC and use it as a simple calculator. It also introduces many concepts that are helpful to know before you start programming in BASIC.

1.1 Entering BASIC

To use BASIC, load CP/M as described in the PCW User Instructions and then:

Type BASIC

After a few seconds, the screen should display a message confirming that BASIC has been selected, similar to the following:

```
Mallard-80 BASIC with Jetsam Version n.nn
© Copyright 1984 Locomotive Software Ltd
All rights reserved
nnnnn free bytes
```

Ok

If this message is not displayed, check your typing and that you have inserted the right disc the right way round.

Once this message is displayed, you have access to all the facilities of BASIC.

1.2 Leaving BASIC

If you want to use a different language or application, you must leave BASIC and return to the computer's operating system. You can either re-boot your computer or use the BASIC SYSTEM command, as follows:

Type SYSTEM

A>

Before you can use BASIC again, you must enter it as described above.

1.3 Using BASIC

Although BASIC is a programming language, it can also be used as a very powerful calculator. Using BASIC as a calculator provides a gentle introduction to many of the fundamental concepts you will need to understand before you can begin to program.

The rest of this chapter introduces you to using BASIC in this way. Using BASIC for programming is described in the next chapter.

1.4 Giving instructions to BASIC

To get BASIC to do anything, you must tell it what to do, by typing instructions for it to obey. An instruction starts with a word that has a special meaning to BASIC, called a 'command', which may be followed by information used by that command, and is finished by pressing `[RETURN]`.

Commands are one of the group of words and symbols that have a special meaning to BASIC called the 'BASIC keywords'. Although you can type them using capitals or small letters, they will always be shown as capitals in this book to help you pick them out (they are also always displayed in capitals whenever you examine your programs).

Commands are rather like the verbs in an English sentence: they indicate an action. There are a wide variety of commands, which will be introduced over the next few chapters. One of the simplest to understand is `PRINT`. This command tells BASIC to display the information contained in the rest of the instruction – to 'print' it on the screen.

The information part of a command is rather like the object in an English sentence: it tells the computer what to apply the command to. In the case of `PRINT`, it tells the computer what to display.

Thus, to get BASIC to print the number 42 on the screen:

Type `PRINT 42 [RETURN]`
 
 command information

Note that BASIC does not try to obey the instruction until you complete it by pressing `[RETURN]`; until then, you are free to alter what you have typed, by deleting characters with the character delete key and retyping.

You should note from the outset that like most other computer facilities, BASIC will only recognise commands if you spell them correctly. Misspelt commands will

normally produce a message from BASIC such as 'Syntax Error'. If you get an unexpected message of this sort, for now just check your typing and try again. In particular, make sure that you leave a space between the command and any information following it.

Another useful and simple command is DIR, which behaves almost exactly as the operating system command of the same name. For instance, to list the names of all files on the currently selected disc:

Type DIR RETURN

1.5 Types of information

The information part of an instruction can take a wide variety of forms, as follows.

1.5.1 Fixed information (constants)

The simplest type of information to understand is that already used in the PRINT example above.

PRINT 42 RETURN
command constant (number)

In this, a number has been specified by typing it in the instruction. This type of information is called a 'constant', because its value is fixed – obvious from the instruction. Constants can either be numbers (such as 42, 0.0000002, -1111111111) or 'text strings' (such as "Hello", "Returns for tax year 87/88", "42"):

Type PRINT "Hello, gentle reader" RETURN
command constant (text string)

Note that text strings must be surrounded by the double quote character (").

1.5.2 Varying information (variables)

A second type of information is called a 'variable'. A variable is a name for an item of information rather than the information itself; when BASIC obeys an instruction containing a variable, it substitutes the value that it has stored for that variable (given by a previous instruction) for the variable name.

But why should you use a variable rather than put the constant value you want in the instruction? By using a variable, you make the instruction versatile – by changing the value of the variable, you can use the same instruction over and over, rather than have a different instruction for each value.

Chapter 1: Starting with BASIC

There are different types of variable to store different types of information. The two main types are 'numeric' (for numbers) and 'string' (for text strings). Numeric variables have names consisting of any of the letters 'A' to 'Z', 'a' to 'z', '0' to '9' and '.'; string variables have similar names but must end with the character '\$'. The first character of a variable name must be a letter, not a number.

Note that although you can use small or capital letters in variable names, they are considered as identical by BASIC. For example, 'dog', 'Dog' and 'DOG' are all the same variable, even though they look different. In this user guide, variable names will be shown using small letters to help you distinguish them from keywords.

Typical variable names are as follows:

```
numeric:  a, count, vat.rate
string:   a$, name$, address$
```

You are free to choose variable names with only a few limitations, the most significant of which is that the variable name must not be the same as a BASIC keyword (as listed in Appendix IV) and may not contain embedded spaces.

The main way to give a value to a variable is to 'assign' the value to it, using the equals sign (=). For example, to give the variable 'vat' a value of 0.15, you could:

Type vat=0.15 RETURN
 variable assigned value

You could then use this variable when you wanted the value 0.15. For example:

Type PRINT vat RETURN
 command variable

Similarly by defining a string variable, such as 'name\$':

Type name\$="Anne Elizabeth Barnard" RETURN
you can print this name easily and quickly, using:

Type PRINT name\$ RETURN
 command variable

The value of a variable is changed, but not fixed by assigning to it: its value can be changed as many times as you want. When the variable is used in an instruction, the value most recently assigned to it will be used.

It may surprise you to see the form that the assignment instruction takes. Where is the command telling BASIC what to do? The answer is that BASIC analyses the instruction, sees the equals sign and interprets it as if there were an invisible

assignment command at the start of the instruction! (In early versions of BASIC, you had to use the assignment command 'LET', but with modern BASICs, although you can use LET, it is no longer necessary.)

1.5.3 Calculated information (expressions)

The third type of information which can be used in an instruction is an 'expression'. An expression is an instruction to BASIC to calculate the information intended. It consists of one or more constants and/or variables, combined with BASIC 'operators' or 'functions'. The operators and functions instruct BASIC how to manipulate the information provided with them to calculate the required information.

Operators Operators are in general single symbols that are used between two items of information, providing a result which is derived from them both.

The simplest expression to understand is one using the familiar operators for addition and subtraction. Try the following instructions using expressions with these operators:

Type PRINT 42+1

and

Type PRINT 43-1

Their meanings and effects should be obvious.

Two other highly useful operators are for multiplication (*) and division (/). While the symbols used may be unfamiliar, their actions are not. Try the following:

Type PRINT 42*2

and

Type PRINT 42/3

These operators work with numeric constants and variables. The only one that works with strings is '+', which adds one string to the end of another. Type the following exactly as shown, but before pressing for the third time, try and predict what will happen.

Type first\$="Annie "

Type last\$="Oakley"

Type PRINT first\$+last\$

Were you right?

Chapter 1: Starting with BASIC

Expressions need not be this simple. For example, to add 22, 44 and 90000032, then take 400311 from the result, you could use three separate instructions:

```
Type    PRINT 22+44 RETURN  
        66  
        Ok
```

```
Type    PRINT 66+90000032 RETURN  
        90000098  
        Ok
```

```
Type    PRINT 90000098-400311 RETURN
```

Or, to save a lot of typing, you can use a single, more complex, expression to get an answer with a single instruction:

```
Type    PRINT 22+44+90000032-400311 RETURN
```

Although there is a limit on the length and complexity of the expressions you can use, you are unlikely to exceed it.

Once an expression contains more than one operator it is not always obvious how it will be evaluated. This is because the order in which the operators are applied can alter the result. For instance, $3+4*5$ could be calculated as:

```
3 + 4 = 7  
    * 5 = 35
```

```
or  
4 * 5 = 20  
    + 3 = 23
```

The BASIC operators and functions have priorities (called the 'operator precedence') which determine the order in which they are applied. For example, * and / are applied to the information to each side of them before + and -, so $3+4*5$ is evaluated as $3+(4*5) = 23$. Operator precedence is described fully in Part II. For now, put round brackets around any part of an expression that you want BASIC to evaluate as a self-contained whole, using the result in the expression. For example $(3+4)*5$ is evaluated as $(7)*5$ while $3+(4*5)$ is evaluated as $3+(20)$.

Functions Functions have names (rather like BASIC commands) and are followed by the information they manipulate, in brackets. A typical function is SQR, which provides a square root:

```
Type    PRINT SQR(121) RETURN  
        11  
        Ok
```

A typical function that works with strings is `LEFT$`, which extracts a specified number of characters from the start of a string. For example, to print the first three letters from `name$` (set up on page 4):

```
Type      PRINT LEFT$(name$, 3) RETURN
          Ann
          Ok
```

(Note how the `LEFT$` function uses two items of information, separated by a comma.)

A third type of function uses information of one type but provides another. For example, to tell how long `name$` is, instead of counting the number of characters yourself, let BASIC do it for you:

```
Type      PRINT LEN(name$) RETURN
          22
          Ok
```

BASIC provides a wide variety of functions such as cosines, logarithms, extracting text from the middle of strings, converting strings to capitals and converting strings to numbers and vice versa. These are described in further detail in Chapter 5 and in Part II. You can even add your own functions to BASIC (using `DEF FN`, as described in Chapter 5).

1.6 Chapter review

In this chapter, you learned how to enter and leave BASIC, and about BASIC instructions, commands and information. You now know how to make BASIC perform simple tasks like calculating and displaying simple sums.

Starting programming with BASIC

This chapter introduces you to fundamental concepts of programming and the simpler tools BASIC provides to help you write and use programs.

2.1 A first program

So far, although you have learned how to use a very limited number of BASIC facilities, you have not written or used a program. This can soon be corrected. Type the following exactly as shown:

```
NEW [RETURN]  
10 PRINT 42 [RETURN]
```

You have now written your first program!

The first line is the command NEW which instructs BASIC to forget any program it might have stored, ready for you to construct a new program from scratch.

The second line is your program. It consists of an instruction (the very first one you met, above), preceded by a number, known as a 'line number'. The line number does two main things: it tells BASIC to store the instruction rather than obey it directly, and gives the stored instruction a label, which can be used for a variety of purposes.


line number instruction

That is all a program is: one or more BASIC instructions typed on separate lines, each of which is preceded by a (different) number.

So how is this simple program different from the similar instruction? Programs have two extra properties:

- programs can be used over and over again with a minimum of effort
- programs can be stored on a disc for use later, even after the computer has been turned off

Chapter 2: Starting programming with BASIC

To make BASIC obey the program you have typed, you use the command RUN:

```
Type    RUN 
        42
        Ok
```

(If you get some other message, there is an error in your program. Start again with NEW etc and be more careful with your typing!)

To store your program on disc, use the command SAVE. Programs are stored by your computer's operating system as named files; you specify the name to use by giving a string after SAVE:

```
Type    SAVE "NUMPROG" 
        Ok
```

The file name should not include a '.' unless you want to specify a file type.

To use the program from the disc, you can make BASIC read it from the disc using the command LOAD, then run it:

```
Type    LOAD "NUMPROG" 
        Ok
Type    RUN 
        42
        Ok
```

Alternatively, you can load and run it with one instruction, by using the command RUN with the program name:

```
Type    RUN "NUMPROG" 
        42
        Ok
```

Note: The program that you load doesn't have to have been typed in within BASIC. It can equally well have been prepared using a word processor or some other text editor and then stored as an ASCII file (ie. as a simple text file).

Once the program has been read from the disc, you can use it over and over again by using RUN without a program name. You do not have to read it again each time.

(If you are a real sceptic you can prove that this version of RUN really works, by turning the computer off then on again, restarting BASIC and only then using the RUN command. This level of scepticism could prove rather inconvenient, though!)

If you use a command such as DIR to examine the disc, you will notice that BASIC has stored the program as a file with the name specified with the SAVE command plus the filetype '.BAS'. This filetype identifies the file as a BASIC program. You do not need to include this filetype when creating or using a program with SAVE, LOAD, RUN (or CHAIN), but you can include a filetype if you want to specify one other than '.BAS'.

2.2 More complex programs

If a program consists of only one line (like the example above), the line number is almost irrelevant. But as soon as you add more lines, the line numbers assume great importance: if the line number of a new line is the same as an existing line, the new line entirely replaces the old one; if the line number is different, the new line is added so that the program consists of lines in order of ascending line number.

For example, type the following four lines exactly as shown:

```
NEW   
30 PRINT 64*64   
10 PRINT "64 times 64"   
20 PRINT "is:" 
```

To see how the program is stored by the computer, use the command LIST:

Type LIST

This displays:

```
10 PRINT "64 times 64"  
20 PRINT "is:"  
30 PRINT 64*64  
Ok
```

As you can see, despite the order in which you typed the lines, BASIC has stored them in the order of their line numbers. This is not just for show: it indicates the order in which the lines will be obeyed when the program is run. To check, run the program and see the results:

Type RUN
64 times 64
is
4096
Ok

BASIC even provides a command to supply line numbers for you automatically when typing in a new program. This command is AUTO:

AUTO <input type="button" value="RETURN"/>	generates line numbers starting at 10, increasing by 10 each time you press <input type="button" value="RETURN"/>
AUTO 620 <input type="button" value="RETURN"/>	generates line numbers starting at the specified start (620), increasing by 10 each time you press <input type="button" value="RETURN"/>
AUTO 100,20 <input type="button" value="RETURN"/>	generates line numbers starting at the first number (100), increasing by the second number (20) each time you press <input type="button" value="RETURN"/>

Chapter 2: Starting programming with BASIC

To turn off AUTO once it has started, type the special character combination called Control-C. Control-C (and other similar Control-*letter* combinations) is typed by holding down the **ALT** key and pressing the given letter key.

In the examples given in this guide, program lines usually contain only a single instruction. Program lines can however have more than one instruction on them, as long as they are separated by colons. Thus, the preceding example program could be written all on one program line as:

```
10 PRINT "64 times 64": PRINT "is:": PRINT 64*64
```

line number first instruction second instruction third instruction

There are occasions when this ability is very useful, but for now you are advised to stick to a single instruction per line, because this makes it easier to change programs, see how they work and trace the cause of the problem when they don't!

2.3 Changing a program

If you are not happy with a program, one way to 'change' it is to wipe it out using NEW and start all over again. While this is perfectly sensible for very short programs, it is a very inefficient way of making small changes to large programs.

BASIC provides a number of commands to make developing programs easier than this:

- To display the program on the screen, use LIST as described above.

If the program is too long to fit onto the screen in one go, you can use LIST and press Control-S to 'freeze' the display when the part that you want to see is on the screen, then press Control-S again to unfreeze the display and continue listing or Control-C to unfreeze the display but stop the listing.

Alternatively, if you know the line numbers of a part of the program you want to display, you can specify them with LIST, as a single number or range, as follows:

LIST 120 RETURN	lists a single line of the program
LIST -80 RETURN	lists from the start of the program up to and including the specified line
LIST 400- RETURN	lists from the specified line to the end of the program
LIST 420-440 RETURN	lists the lines in the specified range

Alternatively, you can list the program on the printer, using LLIST. This behaves exactly as LIST, except that the lines selected are printed on the printer, not on the screen.

- To delete a single line, simply type its line number followed by `RETURN`.

For example, to delete line 400:

Type 400 `RETURN`

To delete a range of lines, use the command `DELETE` with a line or range of lines specified as for `LIST` above.

- To replace an existing line with a new version, type the new line with the line number of the existing line.

- To tidy up the line numbers in the program so that the gaps between them are all the same, use `RENUM`:

Type `RENUM` `RETURN`

This command changes the number of the first line in the program to 10, the next to 20, the next to 30, etc. References to these line numbers within the program are also adjusted accordingly. (`RENUM` is more versatile than this. For example, `RENUM 100` will change the first line number to 100, the second to 110 and so on. Refer to the `RENUM` keyword in Part II for further details.)

- The final facility to help you modify programs is the command `EDIT`. This allows you to modify a specified existing program line.

For example, to edit line 100:

Type: `EDIT 100` `RETURN`

The `EDIT` command displays the current contents of the specified line and provides the following facilities for changing that line:

- movement of the cursor to any position on the program line
- movement of the cursor to a specific character
- deletion of characters before and after the cursor
- deletion of all characters from the cursor up to a specific character
- either insertion of new characters or direct replacement of existing characters (overstrike mode)

These actions can be carried out either by using BASIC's Command editor (by issuing some special editing commands, given in Appendix VI) or, more easily, by using its Screen-based editor. This changes the line 'before your eyes' in response to your pressing particular keys on the keyboard. All the instructions in this part of the BASIC manual assume that you are using the Screen-based editor.

The keystrokes to use on the PCW are as follows:

- `←` and `→` to move the cursor along the line
- `↑` and `↓` to move the cursor between different screen lines where the program line is displayed on more than one screen line
- `FIND character` to move the cursor forward to the specified character
- `← DEL` to delete the character before the cursor
- `DEL →` to delete the character under the cursor
- `CUT character` to delete everything between the cursor and the specified character
- `CUT RETURN` to delete from the cursor to the end of the line
- `IB` to swap from insert mode to overstrike mode or back again
- `STOP` to abandon all the changes that have been made to the program line
- `RETURN` to finish editing the line

To demonstrate how you would use the Screen-based editor, suppose you had typed `20 PRNT` instead of `20 PRINT`. To edit this line, type:

```
EDIT 20 RETURN
20 PRNT
```

Move the cursor to the N by pressing `→` twice

```
20 PRNT
```

Then type I, giving you:

```
20 PRINT
```

Finally press `RETURN` to store the new version.

Note: You can also use the facilities of EDIT on the line you have just typed (whether a command or a program line) by typing `ALT A`.

2.4 Chapter review

In this chapter you have learned how to create programs from the commands and using the types of information introduced in the previous chapter. You have also seen how to store programs on a disc and re-use them, and how to examine and change a program.

Real programming

3.1 Designing programs

As described in the previous chapter, a computer program is simply a collection of numbered instructions you give to the computer, to tell it how to perform a task. Programming is the collection of activities needed to produce a computer program.

Developing even the simplest program will involve you in a number of different tasks. A typical breakdown of activities in the programming of a fairly serious application might be as follows:

- deciding what you want the program to achieve (analysis)
- deciding how the program should achieve what you want (design)
- preparing the program (coding)
- checking that the program works correctly (testing)
- writing instructions for the user (documentation)
- correcting, changing and improving the program (maintenance)

These tasks will be re-examined in great detail in later chapters. For now, the most important thing to realise is that you should design a program before beginning to write it.

To design a program, you must first analyse the task you want the computer to perform into its constituent sub-tasks. When you understand what each sub-task must do, you translate each into a specification of the steps that the computer must take to perform the whole task, expressed using the words and concepts that the programming language provides.

All programs take information, manipulate it in some way, then use the manipulated information to produce results. One of the simplest ways of breaking down a task, then, is to consider it as these three stages:

- obtaining the information
- manipulating the information
- producing the results

Typically, information is typed by the user or read from the program or a disc; is manipulated to produce totals, averages, sorted lists, etc; and is then printed on the screen or on paper as numbers, text, bar charts etc or stored on a disc for later use by this or other programs.

3.2 Getting information

When a program is run, the information that it uses can come from three main sources: from within the program, from the user typing it, or from files stored on discs. Each of these will now be described in turn.

3.2.1 Information storage in the program

If the information that a program needs is fixed, the simplest way to provide it is within the program itself. The simplest way is by using constants, either in instructions:

```
330 PRINT "***SUB-TOTALS***"
```

or by assigning the constant to a variable and using that instead:

```
330 title$ = "***SUB-TOTALS***"  
340 PRINT title$
```

You have already used this method in the example programs and will rarely write a program that does not use it to some extent. Using a variable to hold a constant value in this way has several advantages over using the constant itself:

- if the constant is a long string or number, you need only type it (and get it right!) once, then use the shorter and easier-to-remember variable name throughout the program
- if you assign the constants to variables all in one place, the program is easier to change. For example, if you use the variable `client$` to hold the client name, which is printed in various places in the program, you can easily adapt the program for a different client by assigning a new client name to `client$`, rather than having to search through the program for each mention of the name so you can change it
- variable names can be chosen to make the program easier to understand. For instance, `PRINT value*vat.rate` is rather more obvious than `PRINT value*0.15`

If you use lists of constants in a program, instead of assigning each in turn, you can use a pair of keywords called `DATA` and `READ`. `DATA` is followed by a list of string or numeric values, separated by commas. `READ` is followed by a variable (or list of variables separated by commas) to which it assigns constants in turn from the `DATA` list.

For example, if you wanted to store the number of days in each calendar month, you could use the following DATA statement:

```
100 DATA 31,28,31,30,31,30,31,31,30,31,30,31
```

Then use the following READ instructions to assign the information:

```
110 READ jan.days
120 READ feb.days
130 READ mar.days
etc
```

To save typing, you could include several variables in the READ instruction:

```
110 READ jan.days,feb.days,mar.days
```

READ and DATA are particularly useful for assigning constants to a special type of variable called arrays; see Section 3.3.7, below.

You may have any number of DATA statements in a program. BASIC treats them as if they were a single list of constants, starting with the DATA statement with the lowest line number. Each time READ is used to read a value, the next one is read from the list. If you try to READ more values than are in the list, an error is generated.

A further keyword, RESTORE, can be used to alter the order in which values are read. For information on this and other ways to use DATA and READ, see Part II.

3.2.2 Getting information from the user

The programs that you have so far learned to write have had a major limitation: they each have a single, fixed purpose. If you want one of these programs to do something even slightly different, you have to change the program!

For example, say you run a (very!) simple retail operation, selling a single product and want a program to tell you how much to charge for it to make a 20% profit, the following program would do:

```
10 cost=10
20 profit=0.2
30 PRINT "The price to charge is ";cost+cost*profit
```

Type this program in and run it. (Don't forget to type NEW RETURN first. Use AUTO if you want.)

The program works well, but is rather limited: if the item price changes, you must retype line number 10; if the required profit changes, you must retype line number 20. This is rather tedious and somewhat error-prone. (Try it and see.) It also means

that only someone who knows how to modify the program can use it to calculate the price when the cost or profit margin changes.

The real problem is with lines 10 and 20, because they contain information that is **constant**, while you want the program to cater for information that **varies**. BASIC provides a solution, in the form of a command called INPUT, which asks for information **when the program is run** then assigns it to specified variables.

To see it in action, change line 10 to the following:

```
10 INPUT cost
```

When you run the program now, it displays a '?' and waits. The program is waiting for you to type a number, followed by `[RETURN]`. It will then assign the number to the variable 'cost'. Type a number and press `[RETURN]`. Run the program again, typing a different number.

The changed program is now:

```
10 INPUT cost
    changed
20 profit=0.2
30 PRINT "The price to charge is ";cost+cost*profit
```

INPUT can also be used to assign text to a string variable, by simply using a string variable in the instruction (such as `INPUT name$`). There are certain limitations on the text that this can handle (it does not, for example, cope with commas as you would expect), but these can be overcome (see `LINE INPUT` in Part II).

INPUT has successfully made your program a bit more versatile, but in its new form, the program is still a little daunting to use. When you run it, it is not obvious just what you should do when the '?' prompt is displayed; the program should give a little help. You have already met a command that can be used here: PRINT with a text string.

Modify the program to display a message (prompt) telling the user what to do when they see the '?'. Try the modified program and see if this helps.

Here is one solution:

```
5 PRINT "Type the item cost price, then press RETURN"
    prompt (text string)
10 INPUT cost
20 profit=0.2
30 PRINT "The price to charge is ";cost+cost*profit
```

Did you remember to put the prompt before line 10? The particular text chosen for a prompt is totally irrelevant to how the program works, but is VERY important to the people who use the program. Carefully worded prompts make a program much easier to use.

Because INPUT often needs a prompt to explain to users what they should type, BASIC allows you to include a prompt in the INPUT instruction. Using this, lines 5 and 10 can be combined, as follows:

```
10 INPUT "Type the item cost price; then press RETURN", cost
```

prompt *variable*

```
20 profit=0.2
```

```
30 PRINT "The price to charge is ";cost+cost*profit
```

Note that a prompt used in an INPUT instruction must be a constant text string, not a string variable. (Can you think why? If it were a variable, INPUT would try to assign a value to it, not display it as a prompt!) Also the prompt must come before the variable, not after it.

Note the comma after the text string. This suppresses the '?' prompt that BASIC would normally display after the text string. If you want the '?' displayed, use a semicolon instead of the comma.

Save the program on your disc, with the name "MARKUP". We will return to it later on.

Modify the program to prompt for a profit value too, run it and check that it works correctly.

BASIC provides another method of getting information typed by the user, in the form of the INKEY\$ function. INKEY\$ quickly examines the keyboard to see if a key has been pressed and either returns it as a single character string, or returns a null string if no key is pressed. An instruction using INKEY\$ takes the form:

```
variable$ = INKEY$
```

Since INKEY\$ does not wait for the user to press RETURN (or any other key), it is used quite differently to INPUT. It has two main uses: to check on what the user has typed while carrying on with some other task, and to provide a quick and easy single-character input.

Unfortunately, both uses require BASIC commands that have not been introduced yet. To see INKEY\$ in action, you will have to wait for the next chapter!

3.2.3 Getting information from disc

The third main source of information is from data files stored on discs. Data files are created using BASIC, to hold numeric and string information, rather like variables. However, unlike variables, the information in files is preserved until you deliberately change it: it is not lost when you do things like change programs or switch the computer off.

Using data files is described in detail in Chapters 6 and 7.

3.3 How information is stored

This section describes the types of variables and information that BASIC can use.

3.3.1 Choosing variable names

BASIC leaves the choice of variable name largely up to the programmer. The main options (and restrictions) are as follows:

- use any of the letters 'A' to 'Z' or 'a' to 'z' (capital and small letters are considered equivalent)
- use '.' at any position other than the first character
- use any of the numbers '0' to '9' at any position other than the first character
- use % ! # or \$ only as the last character, and then only when you intend their special meaning (see variable types, below)
- do not use variable names which are identical to a BASIC keyword listed in Appendix IV

Finally, remember that the case (capitals/small letters) of variables is ignored. (The variable `name$` is the same as `Name$`, `NAME$`, etc. etc.)

Within these limitations, you should choose variable names so that they indicate what the variable is being used for. For instance, if a variable is used to maintain a trial balance, use the variable name `'balance'` or even `'trial.balance'`, not something completely obscure like `'zz'`. This will make it far easier for you to understand your programs when you come back to correct or improve them later on.

You should however keep variable names reasonably short, to keep your programs compact and reduce the number of typing errors you will make. Try typing

```
trial.balance.carried.forward
```

```
      = trial.balance.carried.forward + 1
```

a few times and I think you'll agree!

3.3.2 The short life of a variable

Variables are by their very nature rather flighty, transient things. The current value of a variable is lost whenever you:

- RUN a program
- LOAD a program
- leave BASIC

If you try to use a variable for which the value has been lost, it will give 0 (for a numeric variable) or the null string "" (for a string variable).

There are several ways to preserve the information contained in variables so that you can use them later, but for now just remember when writing a program to assign the required values to variables before you use them; don't assume they will be preserved from the last time you used a program: **they won't**.

3.3.3 Numbers

When typing numbers for BASIC to use, you can use a variety of forms, as follows.

The most familiar numbers are what are known as 'unscaled numbers': integers and decimal numbers, such as -44444, 0, 3.111, 3333333333.77702. This is the form of numbers we see and use every day and will probably be most useful for your programming. Note however that you must not include commas or spaces in these numbers.

Probably less familiar (unless you have a scientific background) are 'scaled numbers' (otherwise known as scientific format numbers), in which the number is expressed as a decimal number multiplied by the number 10 raised to a positive or negative number. They take the form *n.nnnE[sign]nn*, or *n.nnnD[sign]nn*. Scaled numbers are mainly useful for expressing very large and very small numbers. (The E and D portions of the number dictate the accuracy with which BASIC stores the number.)

Finally, BASIC provides 'based numbers' to let you express numbers in octal (to base 8) or hexadecimal (to base 16). Octal numbers are preceded by &O; hexadecimal numbers by &H. Based numbers will mainly be of interest if you are used to low level programming with minicomputers or microcomputers.

Typical numbers in each of these representations are as follows:

3.3.5 Text strings

Text strings are sequences of zero or more characters, surrounded by double quotes ("). Typical text strings are:

```
" "  
"and did those feet "  
"Phillip.P.Jones"  
"23"
```

Note that the last is only a string because it is enclosed in quotes; without the quotes it would of course be a number.

What is actually held in the string is not the characters themselves but the 8-bit codes used to represent these characters. These codes are combinations of 0s and 1s in the range 00000000 to 11111111, but are most easily thought of either as hexadecimal numbers in the range &H00...&HFF or as decimal numbers in the range 0...255. The number corresponding to a character is known as the character's internal value.

A consequence of this is that when you print a text string either on the screen or on a printer, BASIC sends the codes for the characters to the output device and it is up to the software controlling the device to translate these codes back into characters. However, the code used for each character depends on the software you are using and your printer, in particular, may not use the same set of codes as your computer – especially as ASCII, the acknowledged standard set of codes, only defines the characters associated with codes in the range &H00...&H7F. So when you go outside the range of characters in the ASCII set (A...Z, a..z, 0...9 and the standard punctuation marks) you can find yourself entering one character and printing another!

Strings can contain any 'character' code between 0 and 255 but not all codes can be entered by typing "*character*": in particular you cannot use this method to enter some of the codes because these are used to represent actions, such as starting a new line or a new page, rather than printable characters. However, there is a BASIC function, CHR\$, which can be used to enter any code. You just need to know the internal value of the code. We will see how CHR\$ is used later in this chapter when we use these 'Control' codes to control the screen and the printer.

3.3.6 String variables

String variables are distinguished from other types by a suffix of \$. The maximum number of characters that can be stored in a string variable is 255; the minimum is zero (the null string, ""). Characters with internal values anywhere between 0 and 255 can be stored in string variables.

3.3.7 Organising variables (arrays)

The variables described up until now have all had quite independent, separate, existences. While this is entirely satisfactory for programs that simply take information, manipulate it and print the results, there are times when you will want to hold lists or tables of information in variables, preserving the relationship between them.

One way of doing this is to use related variable names. For example, to store the names of record albums, you could use the string variables 'album1\$', 'album2\$', 'album3\$' etc.

While this method of naming makes the relationship obvious to you, BASIC entirely misses the point – it sees no ordering or grouping in these names. For instance, to print the name of album 2, you need the instruction:

```
PRINT album2$
```

To print the name of album 3, you need a different instruction:

```
PRINT album3$
```

And so on, for each name. Even more seriously, there is no way (with the commands described so far) to construct a program to perform simple, useful tasks like printing the album name from its number.

Luckily, BASIC provides a way to solve these problems by organising variables into an arrangement called an 'array'. A variable in an array is called (naturally enough) an 'array variable' or 'array element'.

An array consists of a group of variables with the same name and type. In the simplest form of array, the variables are organised into a list, being distinguished by a numeric value in brackets added to the end of the variable name. This value is known as the 'array index' and is 0 for the first element, 1 for the next one and so on (although you can make BASIC give the first element an index of 1; see `OPTION BASE`).

Arrays can be made using any of the types of variable described above. For example, while `album$` is a string variable, `album$(6)` is a (string) array variable and `album(3)` is a (numeric) array variable. Note that all these variables are distinct.

Array variables can be used exactly as the equivalent (non-array) variable type, but their main use is as indicated above: in processing lists or tables of information. For example, to solve the problem posed above, the following program will suffice:

```
10 album$(0)="Dark side of the moon"
20 album$(1)="Vivaldi's four seasons"
30 album$(2)="A bridge over troubled water"
40 INPUT "What number album do you want a title
                                     for";number
50 PRINT album$(number)
```

Lines 10 to 30 give values to the first three elements in the array album\$(). Line 40 gets the album number wanted by the user. Line 50 prints the string stored in the corresponding array variable.

If you are going to use an array with more than ten elements, or want to use fewer elements and not waste memory, you should 'dimension' the array before using it, with the command DIM. DIM is followed by the array name and the maximum subscript to be used in brackets, for example:

```
10 DIM album$(3)
```

Arrays can have many dimensions, not just the single one of album\$() above. For instance, if you wanted to record the temperature at a number of points within a room, you might use 'temp(x,y,z)' where x, y and z are the co-ordinates of the point within the room, chosen so that no two points will have the same x, y and z co-ordinates. If you wanted to record how this information changed with time, you could use 'temp(x,y,z,t)', where t is the time scale, and so on. Before you can use an array with more than one dimension, you must dimension the array using DIM (for example, DIM temp(10,10,15)).

Note that an array cannot be used as a variable, only the elements in it. An array is just a way of organising variables for your convenience.

The quickest and easiest way of setting up an array from constants is to use the keywords DATA and READ, introduced above. Contrast the following program fragments, both of which store the days in the month in an array called 'days'. The first uses DATA, READ and a FOR...NEXT loop to produce code which is compact and easy to understand; the second does not. The FOR...NEXT loop will be explained in full in Section 4.2.

```
10 DIM days(12)
20 DATA 31,28,31,30,31,30,31,31,30,31,30,31
30 FOR month = 1 TO 12
40   READ days(month)
50 NEXT
```

```
10 DIM days(12)
20 days(1) = 31
30 days(2) = 28
40 days(3) = 31
50 days(4) = 30
60 days(5) = 31
70 days(6) = 30
80 days(7) = 31
90 days(8) = 31
100 days(9) = 30
110 days(10) = 31
120 days(11) = 30
130 days(12) = 31
```

3.3.8 Choosing variable types

The choice of variable type can be broken into two levels: numeric or string, and integer or single or double precision.

String variables must be used if you want to store text (numeric variables cannot be used for this), but can also be used to store numbers if they are not going to be manipulated as numbers.

Integer variables should be used whenever you don't need to store fractional information and the number is in the permitted range (-32768 to $+32767$), because integer variables produce smaller, faster programs.

Single precision variables will probably be sufficient for most other uses. As a rough guide, you should only need to use double precision variables if you want to preserve more than six significant digits in your calculations. Double precision variables take up more memory and considerably increase the time taken to perform calculations.

An awkward point to bear in mind is that while we think and calculate in decimal, most computers do so in binary, converting decimal numbers to binary for storage and calculation, converting the binary back to decimal for display. Fractional numbers which can be expressed exactly in decimal, can't necessarily be expressed exactly in binary and as a result the answers to calculations may be less accurate than you expect. For example, the result of a simple calculation might be 0.9999998 rather than 1.

This 'inaccuracy' is a direct result of expressing floating point numbers in binary and it affects all programs that carry out floating point arithmetic. Another effect is

that numbers that appear the same when you print them aren't necessarily identical, because the displayed numbers will probably have been 'rounded' to some number of decimal places. In practice, you should always round any calculated result before using it, with the BASIC function `ROUND` (see Section 5.3.2).

Array variables are generally useful when the information being stored is obviously ordered, into lists or tables, and you want to preserve that ordering. A typical example of a table of data is an address list, which could be stored using separate arrays for the name, address and telephone number, linking them for each individual by a shared index number (eg. `name$(22)` plus `address$(22)` plus `phone$(22)` as the entry for one individual).

3.4 Outputting the results

The purpose of any program is to produce tangible, usable results. This is described as the program's Output.

There are two main types of output: visible (human-readable) output on the screen or on a printer and 'invisible' (machine-readable) output which is usually stored on disc prior to being processed by another program. What we shall look at here is visible output. Output to disc is described in Chapter 6.

3.4.1 Output to the screen

The command used to send output to the screen is `PRINT`. This command has already been used extensively in previous examples but in a rather limited form. All we have done is print single items, either on separate lines or one after the other along the same line.

The display these simple commands produced has often been rather untidy, with information split between lines and with similar information on different lines not lining up vertically. In fact, `PRINT` instructions can give you much greater control over where each piece of information is displayed.

The simplest `PRINT` instruction consists of just the keyword `PRINT`. This moves the cursor to the start of the next line and is most frequently used to produce a blank line. More complex `PRINT` instructions follow the keyword `PRINT` by combinations of items to print and instructions that control their printing.

The items to print must be separated by commas or semicolons. If the two items are separated by a semicolon, the second item is printed immediately after the first, without leaving a space. However, if the items are separated by a comma, the second item is positioned at the start of the next print zone.

Chapter 3: Real programming

Print zones have an effect rather like tab stops on a typewriter: they enable you to put your output into columns. Each print zone gives you a separate column.

To see the effect of the print zones, type and run the following program:

```
10 PRINT "Print zone number"
20 PRINT "1", "2", "3", "4", "5", "6"
```

Initially the zones are 15 characters wide (effectively, giving you a tab stop every 15 characters across the screen), but you can change the width of the zones to n characters by using a `ZONE n` command. For example, you could make each column 10 characters wide by having a program line containing the instruction `ZONE 10`.

The items to be printed may be numeric or string constants, variables or expressions, or the 'print functions' `SPC(n)` and `TAB(n)`. Printing `SPC(n)` prints n spaces; printing `TAB(n)` moves the cursor to column n on the current line, or to column n on the next line if the cursor is to the right of column n . So, for example, you could print three numbers separated by six spaces with an instruction like:

```
PRINT number.1;SPC(6);number.2;SPC(6);number.3
```

After all the items in the print instruction have been printed, the cursor normally moves to the start of the next line. However, if the instruction ends in a comma, semicolon, `SPC(n)` or `TAB(n)`, the cursor will stay where it is. To see this effect, run the following program and note how the semicolon 'joins up' the output from lines 30 and 40.

```
10 PRINT "Different"
20 PRINT "lines"
30 PRINT "The same ";
40 PRINT "line"
```

Controlling the format of the individual items

The PRINT commands we have used so far have displayed your list of print items 'free format' – that is, according to a simple set of standard rules.

If you are not happy with the way PRINT displays print items, you can give your own specification of how they should be shown. To do this, you include the keyword `USING` in your PRINT statement, followed by a 'format template'. Format templates give you a wide range of options in presenting text and particularly numbers. Full details of format templates are given in Part II (Chapter 6), but the following example should give you an idea of the flexibility provided.

Suppose you are handling amounts of money in a program. Naturally, you will want to tabulate these neatly. At first glance, you might think that all you need to do to produce a neat table is to use a TAB function, as in line 50 below:


```
10 sum(1)=1.24
20 sum(2)=0.333
30 sum(3)=1444
40 FOR count = 1 TO 3
50 PRINT TAB(40); sum(count)
60 NEXT
```

Type this program and then run it. As you will see, the figures line up but by their first character – not by the decimal point as you would like.

You can make the decimal points line up if you change line 50 to:

```
50 PRINT TAB(40); USING "###.###.##"; sum(count)
```

The "###.###.##" is the format template that is to be used for `sum(count)`. The pattern of `#s` specifies that each number should be displayed as six figures to the left of the decimal point (if necessary, padded out with spaces) followed by two to the right of the decimal place (with the second decimal place rounded). The comma specifies that the figures to the left of the decimal should be grouped in threes as shown in the template, with the groups separated by commas.

Run the revised program to see its effect.

3.4.2 Output to the printer

Outputting information to the printer uses the `LPRINT` command, rather than the `PRINT` command, but is otherwise just like outputting information to the screen. The instructions used are very similar and all the print functions described above (`TAB`, `SPC`) and the format templates have the same effect.

3.5 Controlling the output device

Print functions and format templates don't represent the only control you have over how your output is presented. You can also control the device itself. In the case of screen output, this means you can choose whereabouts on the screen each print item is placed – and you can clear away information you no longer require. On a printer, as well as moving the print position, you can select different print modes and control such things as the number of characters per inch across the page (the Character pitch) and the distance between one line and the next (the Line Pitch).

The way you control these devices is by sending some special sequences of 8-bit codes to the device. These control sequences are either single characters with internal values in the range 0...31 (Control codes), or short groups of two or three characters, the first of which is the 'Escape' character (internal value 27; also known as `ESC`). The groups of characters are recognised by the device as control sequences and are acted upon accordingly.

A number of programs contain the facilities to send control sequences to the screen and the printer. What we describe here is how you send these control sequences from BASIC.

3.5.1 Controlling the screen

The actual facilities and the codes required depend on the computer you are using and the operating system you are running. However, most systems are designed to provide the same facilities and use the same control sequences as one of the standard computer terminals – in particular, the VT52 and ADM3A terminals.

Details of the codes you should use will be given in your computer's own user guide, under a title like 'Screen handling', 'Monitor control' or 'Terminal characteristics'. This may also tell you which standard terminal is being emulated.

CP/M Plus, the operating system on the PCW, provides broadly the same facilities as and uses the same control sequences as a VT52 terminal. The codes are listed in Appendix III 'Terminal characteristics' in the section of the PCW User Instructions that describes using CP/M.

The best way of showing how to make use of these codes is through an example. Suppose, for instance, that you wanted to:

- Clear the screen entirely
- Write the title 'Screen control' on the top line of the screen
- Then move the cursor to line 15 and column 22, ready for the next piece of information:

Looking at the appendix on 'Terminal Characteristics', you will see that you will need:

- ESC E to clear the screen (or, more precisely, the section of the screen that your program can use but this is normally either the whole screen or all but the bottom line)
- ESC H to move the cursor to its Home position in the top lefthand corner of the screen (ESC E leaves the cursor in its old position)
- ESC Y $r+32$ $c+32$ moves the cursor to a given position, where r is the row number and c is the column number (both counted from the top lefthand corner of the screen)

(The rows referred to here are the lines of the screen and the columns are the character positions across the screen, all characters on the screen being displayed on a rectangular grid, so many lines deep and so many characters wide.)

To get the result you require, you should first send the escape sequence ESC E to clear the screen; follow this with the escape sequence ESC H to position the cursor at the top of the screen; then print the title 'Screen control'; and finally send the escape sequence ESC Y 47 54 to position the cursor at line 15, column 22 (47 is 15+32; 54 is 22+32).

The technique BASIC uses to send control sequences to the screen takes advantage of the fact that both control sequences and text strings are made up of 8-bit codes. So just as you use PRINT statements to send text strings to the screen, so you use PRINT statements to send the control sequences to the screen. The only complication is how you enter a control sequence that is given here as ESC Y 47 54, for example, in your PRINT statement.

The first stage is to understand what ESC Y 47 54 represents. Each part of this control sequence is a separate code. These codes can be written in one of three ways – as the code's 'name', as the character associated with the code (where this is a printable character, like Y), or as the corresponding internal value. In the case of ESC Y 47 54, ESC is the code's name, Y is the character associated with the code and 47 and 54 are internal values. (You can always assume that straightforward numbers are internal values; if the code corresponding to one of the characters 0...9 is required, then it will usually be written as "0"... "9" to make this clear.)

The techniques used to express these codes are simply those used to enter characters into a text string.

- If the code is associated with a printable character, you can express it as "*character*"; so to specify Y, you can put "Y"
- But in general, you can specify it by using the CHR\$ function and quoting its internal value; so for example, to specify 47, you put CHR\$ (47)

Codes with special names don't represent printable characters and you express them using CHR\$ (*value*) where *value* is the code's internal value. For example, ESC has the internal value 27 and so to specify ESC, you put CHR\$ (27) .

You can put the codes of the escape sequence into the PRINT statement as individual print items, separated by semicolons. (Alternatively, you can combine them into a string of characters by using the + operator, described in Section 5.2.2 'Joining strings'.) So the complete sequence ESC Y 47 54 could be sent as:

```
PRINT CHR$ (27) ; "Y" ; CHR$ (47) ; CHR$ (54) ;
```

(The final semicolon is included to prevent the cursor from moving to the start of the next line on the screen after completing this instruction – as described above.)

As we emphasised earlier, using variable names rather than straightforward values

Chapter 3: Real programming

makes a program easier to understand and so we would recommend defining any escape sequences and control codes you plan to use at the beginning of your program. For the series of actions we suggested above, you might start your program with:

```
10 escape$=CHR$(27)
20 clear$=escape$+"E"
30 home$=escape$+"H"
40 move$=escape$+"Y"
```

(Lines 20..40 use the technique described in Section 5.2.2 to combine ESC with the next character in the escape sequence.)

With these defined, you could then use the following set of PRINT statements to clear the screen, write the title 'Screen control' and move to line 15, column 22:

```
110 PRINT clear$;home$;
120 PRINT "Screen control"
130 lineno = 15
140 column = 22
150 PRINT move$;CHR$(lineno+32);CHR$(column+32);
160 PRINT "New text position";
```

Finding out the control codes and expressing them in PRINT statements is essentially all there is to screen control. However, there are a couple of further complications to do with other features of the way BASIC works.

The first concerns the code with the internal value 9. This is normally treated by BASIC as a TAB character and automatically replaced by the spaces required to take the cursor to the next tab position. Whenever you need to use this special character as a control code or as part of an escape sequence, you should first use the command `OPTION NOT TAB` to suppress tab expansion. (If you want to use tab expansion again later in the program, you can re-enable it with the command `OPTION TAB`.)

The second concerns the way BASIC automatically starts a new line when you reach the righthand side of the screen. It does this by inserting a couple of control codes (CR and LF) at an appropriate point in the list of print items. It is quite possible that these extra codes could be inserted in such a way that your escape sequences no longer have their desired effect.

The solution to this problem is to precede your group of control codes and escape sequences by a program line containing the instruction `WIDTH 255`, which turns off the automatic next-line feature. If, at the end of the sequence of codes, you want restore this feature, you can do this with the instruction `WIDTH n`, where *n* is the number of character positions across the screen. On the PCW, the screen is 90 characters wide and so the instruction you will need is `WIDTH 90`.

Thus the complete series of instructions might be:

```
10 escape$=CHR$(27)
20 clear$=escape$+"E"
30 home$=escape$+"H"
40 move$=escape$+"Y"
100 OPTION NOT TAB : WIDTH 255
110 PRINT clear$;home$;
120 PRINT "Screen control"
130 lineno = 15
140 column = 22
150 PRINT move$;CHR$(lineno+32);CHR$(column+32);
160 PRINT "New text position";
170 OPTION TAB : WIDTH 90
```

Note: The series of codes `move$;CHR$(lineno+32);CHR$(column+32)` can be very neatly packaged into a user-defined function. With such a function set up, you could move anywhere on the screen simply by quoting the line number and the column number. Details of user-defined functions are given in Chapter 4.

3.5.2 Controlling the printer

The codes used to control a printer, like those used to control the screen, are either single Control codes with internal values in the range 0...31 or Escape sequences starting with the special 'ESC' character (internal value 27).

You use these codes for two main types of task:

- To lay out and style text – by preceding the text by the codes needed to give it the required layout and style
- To carry out specific actions such as starting a new page – by embedding the codes in the text where you want the action to happen.

The codes you use depend on the printer you want to control – not on the computer you are using. So the place to look for details of these codes is in the printer's own manual. The codes that control the built-in printer are given in Part III, Appendix II of the PCW User Instructions.

A number of printers use the same control sequences: such printers are described as being compatible. For example, there are a number of daisy-wheel printers that use the same codes as a Diablo 630, while some dot-matrix printers use the same codes as the Epson FX-80. However, these 'standard' codes only control a fairly limited range of printer features, so the printer manufacturer will normally have included additional 'non-standard' codes so that you can use all the printer's facilities. The PCW's built-in daisy-wheel printer in fact has two modes of operation – one in which it uses the same codes as a Diablo 630 and the other in which it uses the same codes as an Epson FX-80. In both these modes, extra control sequences have been included so that you can make full use of the printer's facilities.

The control sequences are sent to the printer by inserting them in LPRINT statements. The codes in each control sequence are expressed either as "character" (when they are associated with printable characters like Y) or as CHR\$(value) where value is the code's internal value – just as they are when you are using codes to control the screen. Indeed, the whole process of converting the printer's codes into LPRINT statements can be directly modelled on the statements used above to control the screen. Again, you need to remember to use OPTION NOT TAB to stop any characters with internal value 9 from being interpreted as a tab and expanded into a number of spaces, and you should use WIDTH LPRINT 255 to stop BASIC's automatic new-line feature affecting printer escape sequences. (Again, restore the new-line feature by using another WIDTH LPRINT command which resets the width of the printer.)

Often, the most difficult part of controlling the printer is in fact in working out the details of the codes you require. So as our example of LPRINT statements containing printer control sequences, we will use first the Diablo 630 codes and then the FX-80 codes that you would use to set Character pitch 12 (ie. 12 characters per inch along a line) and Line pitch 8 (ie. lines 1/8 inch apart down the page) because these are a couple of the more difficult codes to work out.

Example of using Diablo-630 codes

To set the Character Pitch on a Diablo 630 printer you have to set the distance the printhead moves between printing one character and the next. This distance is defined by the Horizontal Movement Index (HMI): if the HMI is n , then the distance moved is $n/120$ inches. Similarly, to set the Line pitch, you have to set the Vertical Movement Index (VMI): if the VMI is n , then the distance between one line and the next is $n/48$ inches. To use Character pitch 12 and Line Pitch 8, you therefore have to set an HMI of 10 (so that the distance moved is $10/120 = 1/12$ inch) and a VMI of 6 (so that the distance between lines is $6/48 = 1/8$ inch).

The escape sequence that sets the HMI to 10 is ESC US 11 (note how the number in the code is one more than the HMI you want to set) and the escape sequence that sets the VMI to 6 is ESC RS 7 (again, note how the number in the code is one more than the VMI you want to set). ESC is the familiar character with internal value 27; US and RS are the names of two more special characters (internal values 31 and 30, respectively); and the numbers are internal values. So you could set Character Pitch 12 and Line Pitch 8 with the following statements:

```
escape$=CHR$(27) : us$=CHR$(31) : rs$=CHR$(30)
hmi=10 : vmi=6
LPRINT escape$;us$;CHR$(hmi+1);: REM set Character Pitch 12
LPRINT escape$;rs$;CHR$(vmi+1);: REM set Line Pitch 8
```

Example of using FX-80 codes

The FX-80 printer uses individual escape sequences to set the different Character pitches and Line pitches. For example, the code that sets Character Pitch 12 is ESC M, whereas both the special character SI and the sequence ESC SI can be used to set Character Pitch 17. Similarly, the codes used to set Line Pitch 6 and Line Pitch 8 are ESC 2 and ESC 0, respectively.

The complication here is that, because 10 Pitch is the initial pitch used on these machines, the code used to set this pitch depends on the pitch that you are using at present. For example, if you are returning to 10 Pitch from 12 Pitch, the code to use is ESC P, but if you are returning to 10 Pitch from 17 Pitch, the code to use is the special character DC2.

The codes used to set Character Pitch 12 and Line Pitch 8 are ESC M and ESC 0 (zero), respectively. So you could use following statements to set this combination:

```
escape$=CHR$(27)
LPRINT escape$;"M"; : REM set Character Pitch 12
LPRINT escape$;"0"; : REM set Line Pitch 8
```

The technique of using Print commands to control the screen and the printer described here isn't used by every version of BASIC you might use. Many have specific commands to carry out simple actions such as clearing the screen or positioning the cursor. These are simpler to use than control sequences but the advantage of the approach used in Mallard BASIC is that the same technique is used whatever action you require. Moreover, special screen and printer commands tend to be only suitable when you are using a particular computer, whereas Mallard BASIC can be used to control the screen and the printer on a very wide range of computer systems.

Building larger programs

Although a program is simply a number of instructions contained in program lines, in many ways it is more than the sum of its parts. One of the major factors affecting the success of a program is the way that you organise the instructions in it to form the program. The organisation of the program (its structure) is the topic of this chapter.

4.1 Compartmentalising a program

Very short programs are usually easy to understand and follow; as long as you know what the individual instructions do, you can figure out what the whole program does – line 10 gets a value, line 20 multiplies it by 3, line 30 prints the result and so on.

As soon as a program gets beyond about ten lines long or starts to use complex instructions, this is no longer the case. A program which is difficult to understand is difficult to write and doubly difficult to change or correct!

The way to make a long or complex program easier to understand is to break it down into smaller chunks and label each chunk with what it does.

To break a program into chunks, decide what separate stages the task consists of and write instructions to complete each stage in turn, but between each stage and the next, enter a program line consisting of just ' on its own.

To indicate what a particular stage in the program does, use REM. This is a strange keyword – it simply tells BASIC to ignore the rest of the line! This allows you to type comments after the REM, usually to explain what is going on.

You will see both methods being used throughout the example programs in the rest of Part 1. To give you a taste, contrast the following programs:

```
10 title$ = "Markup program"
20 markup = 0.5
30 PRINT title$:PRINT
40 INPUT "Product price";price
50 INPUT "Product name";name$
60 markup.price = price*(1+markup)
70 vat = 0.15
80 INPUT "How many";number
90 value = number*markup.price*(1+vat)
100 PRINT "Your stock of ";name$;" should cost ";value
```

```
10 REM ** set up variables **
20 '
30 vat = 0.15
40 markup = 0.50
50 title$ = "Markup program"
60 '
70 REM ** set up screen **
80 '
90 PRINT title$: PRINT
100 '
110 REM ** get input **
120 '
130 INPUT "Product price";price
140 INPUT "Product name";name$
150 INPUT "How many";number
160 '
170 REM ** calculate value including vat and markup **
180 '
190 markup.price = price*(1+markup)
200 value = number*markup.price*(1+vat)
210 '
220 REM ** display value **
230 '
240 PRINT "Your stock of ";name$;" should cost ";value
```

Although the second, neatly structured, program is somewhat longer, it is far easier to understand and check. For example, have you assigned the value to 'vat' before using it? In the first version, you have to hunt for an answer; in the second, all constants are assigned to variables in one place, making it easy to check.

If you are going to write programs of any size or complexity, you should try to write them like the second version. Changing a program like the first will be a real nightmare!

4.2 Sequences and loops

The programs you have written so far have all been executed as a list of instructions, starting at the lowest line number and ending after executing the highest. There are however many ways to change the order in which program lines are executed.

The first of these is the GOTO command, which is followed by a line number: the target line. When this instruction is executed, BASIC continues execution from the target line specified rather than the next line in number sequence.

This may seem a little silly at first glance, but try the effect of adding the following line to the example program named "MARKUP". Load the program which you saved earlier by typing:

```
LOAD "markup"
```

Now add the new line by typing:

```
40 GOTO 10
```

The complete program is now:

```
10 INPUT "Type the item cost; then press RETURN",cost
20 profit=0.2
30 PRINT "The price to charge is ";cost+cost*profit
40 GOTO 10
```

Now run the program. See how much easier it is to use the revised program: if you want to calculate the price to charge for a number of items, instead of having to run the program once for each item, it is ready for you to type the next cost price as soon as it has displayed the charge price for the previous item.

Repeating the same sequence of instructions over and over again like this is called 'looping', with the repeated instructions called a 'loop'.

This revised version of the program using GOTO is all very well, but now the program never stops! The only way to get out of the loop is to press Control C, which immediately stops the BASIC program.

While this approach is quite acceptable in a simple program like this, it does not help you to write programs where you want to leave the loop and carry on executing another part of the program. A far better solution is described below.

Looping is used so frequently that BASIC provides commands to define and execute loops.

The first type of loop starts with the command FOR and ends with the command NEXT. The simplest form of this loop is as follows:

```
FOR counter=start TO finish
  instructions
NEXT
```

Chapter 4: Building larger programs

When it executes the FOR instruction, BASIC sets the numeric variable *counter* to the numeric value *start*. It then checks whether *counter* is less than or equal to *finish*. If it is not, BASIC skips the whole loop. If it is, BASIC obeys the following instructions until it reaches the NEXT command, then adds 1 to the *counter* variable and checks whether its new value is greater than the numeric value *finish*. If not, the instructions inside the loop are executed again, *counter* is increased by 1, then tested against *finish* again. Only when *counter* is greater than *finish* does BASIC finish the loop and continue with the next instruction after the NEXT command.

This type of loop is usually called a 'FOR...NEXT loop', for obvious reasons.

The FOR...NEXT loop is most useful when the number of times the loop should be executed is known at the start of the loop. For example, to print a column of asterisks down the screen:

```
10 INPUT "How many asterisks";asterisks
20 FOR a=1 TO asterisks
30 PRINT "*"
40 NEXT
```

This type of loop is also very convenient if you want to keep a count of the number of times the instructions in the loop have been executed. For example, to program a 'times table':

```
10 INPUT "What number times table do you want";times%
20 FOR number%=1 TO 12
30 PRINT number%*times%
40 NEXT
```

See how the counter variable is used not only to control the loop, but also to generate the output.

4.3 Making decisions

So far, we have seen how the computer can perform calculations, store and retrieve information, even repeat sequences of commands over and over. All very impressive, but not enough to program many tasks; remember the problem with leaving the loop created using GOTO, above?

The vital missing link is the ability to make decisions. Without it, the computer can only function as a powerful calculator; with it, really versatile programs can be written which adjust their behaviour to the information they receive, even (in extreme cases) appearing to be 'intelligent'.

4.3.1 Choosing alternative statements

Decision-making in BASIC is mainly provided by the commands IF and ON. IF will be described first.

IF: IF is followed by a 'logical expression' that BASIC can test, such as 'count is greater than limit'. In its simplest form described here, if BASIC finds that the expression is true, it obeys the instructions following the keyword THEN on the line. If the expression is false, (for instance if count is 4 and limit is 6), these instructions will be skipped over and execution will continue on the next program line.

This simple version of the IF instruction takes the following form:

```
IF logical-expression THEN instructions
   expression tested      executed if true
```

Note that the whole structure of the IF command, the logical expression, THEN and the instructions that depend on it must all be on the same program line. This is one of the instances when you will often need to include several instructions on the same line (by separating them with colons).

In the example program named 'MARKUP', we could decide that to leave the loop, the user will type a cost price of zero. The logical expression to test if cost is equal to zero is 'cost=0'. To leave the loop we could use GOTO to carry on execution with a program line after the loop (line 50). Type the following additional lines:

```
15 IF cost=0 THEN GOTO 50
      expression      executed if true
```

```
50 PRINT "And now for the rest of the program ..."
```

The modified program is now:

```
10 INPUT "Type the item cost, then press RETURN",cost
15 IF cost=0 THEN GOTO 50
20 profit=0.2
30 PRINT "The price to charge is";cost+cost*profit
40 GOTO 10
50 PRINT "And now for the rest of the program ..."
```

Try running the program to check that it works correctly. Try typing positive (or even negative!) cost prices and see that the looping continues. Type a cost price of 0; this should display the message produced by line 50 and end the program.

Chapter 4: Building larger programs

IF can also be followed by the keyword ELSE, which allows you to specify that a group of instructions will only be executed if the expression is not true:

IF *logical-expression* THEN *instructions-1* ELSE *instructions-2*

expression tested *if true* *if false*

For example, consider the following program fragment:

```
90...
100 balance = balance + transaction
110 IF balance<0 THEN PRINT "***You're overdrawn***": PRINT
    "By " ; -balance ELSE PRINT "You're in credit" : PRINT
    "With " ; balance
120 INPUT "What next? (Q-quit, T-transfer, I-invoice)";
                                answer$
130...
```

Line 110 uses IF...THEN...ELSE to print a different message depending on the state of your bank balance:

```
      THEN PRINT "***You're overdrawn***":PRINT "By " ; -balance
      /
IF balance<0
      \
      ELSE PRINT "You're in credit":PRINT "With " ; balance
```

Note that, as for IF...THEN, the whole structure of IF...THEN...ELSE must be on a single program line (though this can flow onto several screen lines).

ON : The ON keyword is followed by a numeric expression, GOTO and a list of line numbers:

ON *numeric-expression* GOTO *line1*, *line2*, *line3*, etc.

The numeric expression is evaluated when the instruction is executed to yield a number *n*. If *n* is between 1 and the number of line numbers in the list, execution continues at the *n*th line number in the list. If *n* is less than 1 or greater than the number of line numbers in the list, execution continues with the next instruction.

ON provides a neat way to select one from a number of choices. It is equivalent to a sequence of IF instructions:

```
IF numeric-expression = 1 THEN GOTO line1
IF numeric-expression = 2 THEN GOTO line2
IF numeric-expression = 3 THEN GOTO line3
etc.
```

(ON is also used with a number of other commands. These are described later.)

4.3.2 The tests

As described above, the *logical-expression* is usually a statement of the relationship between variables and/or constants. This will usually use one of the following 'relational operators':

<code>a < b</code>	true if a is less than b
<code>a <= b</code>	true if a is less than or equal to b
<code>a = b</code>	true if a is equal to b
<code>a <> b</code>	true if a is not equal to b
<code>a >= b</code>	true if a is greater than or equal to b
<code>a > b</code>	true if a is greater than b

Typical logical expressions are:

```
count1 < count2
name$ = ""
key <> pointer%*2+1
file.absent.flag
```

These logical expressions are evaluated by BASIC to produce either 0 (for false) or -1 (for true). You may also use a numeric constant or variable as a logical expression, which BASIC will consider as 'true' unless it has a value of 0.

Logical expressions can also be combined and modified using logical operators. The most useful are NOT, AND and OR.

NOT is used to invert the sense of a logical expression, so if 'a > 6' evaluates to 'true', 'NOT (a > 6)' evaluates to 'false'.

OR is used to combine two logical expressions; the result is 'true' if either (or both) are 'true'. Thus, the result of '(1 = 2) OR (3 = 3)' is 'true' because while the first expression is 'false', the second is 'true'.

AND is used to combine two logical expressions; the result is 'true' only if both are 'true'. Thus, the result of '(1 = 2) AND (3 = 3)' is 'false' because while the second expression is 'true', the first is 'false'.

(The logical operators actually perform what are known as bitwise operations on the integer values of -1 and 0 BASIC uses to represent true and false. See Section 5.1.5)

4.3.3 Controlling a loop

IF causes BASIC to execute a group of instructions once if a condition is true. Another command, WHILE, is provided to make BASIC execute a loop while a condition is true.

The WHILE loop is started by the command WHILE and ended by the command WEND. It takes the form:

```
WHILE logical-expression  
    instructions  
WEND
```

When it executes the WHILE instruction, BASIC evaluates the *logical-expression* and if it is true, executes the instructions in the loop up to the WEND command, then evaluates the *logical-expression* again as if entering the loop for the first time. BASIC only skips the loop and continues with the instruction after the WEND command when the *logical-expression* is found to be false.

The WHILE loop is ideally suited to situations where the number of times a loop should be executed is not known when it is started. As the WHILE logical expression is tested again each time the WEND is executed, all the programmer has to do is make sure that this expression evaluates to false when the looping should stop.

4.4 Stopping the program

Left to its own devices, a BASIC program only stops when the last program line has been executed. You can also stop it at any time by pressing Control-C – a sort of emergency stop!

BASIC provides two commands to let the programmer make the program stop itself under different circumstances: END and STOP

END stops the program just like it stops when it runs out of program lines. END is mainly used to stop the program when it has finished its intended tasks but there are more lines in the program.

STOP stops the program and displays the number of the program line containing the STOP command. STOP is mainly used when testing a program that works incorrectly, to provide an easy way to check which parts of the program are being executed. (The command CONT can be used to make BASIC resume execution from where it stopped.)

4.5 Organising the program

4.5.1 Review of program structure

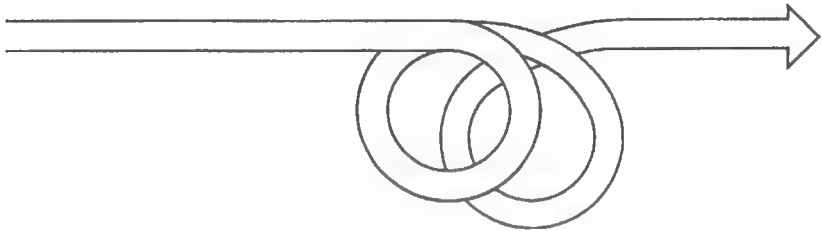
You have already met several different ways of controlling the order in which lines are executed. The first examples above executed each program line just once, in numerical order, rather like a straight road, running from start to finish:

A 'straight line' program:



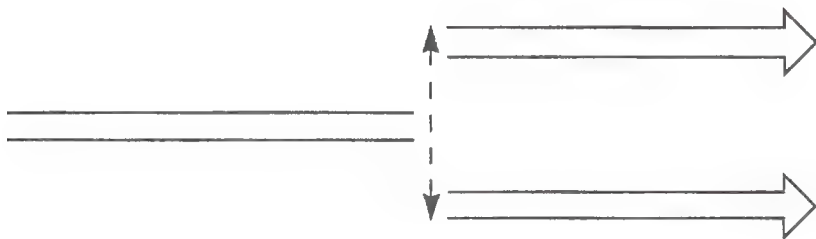
Next came programs that looped: the program still ran from start to finish, but along the way parts were repeated, rather like a road with a roundabout and a driver who couldn't decide which exit to take:

A program with looping:



Then came programs that made decisions; there was a fork in the road and the program took the left or right path depending on information it had been given or calculated when run:

A program with decision-making



These different structures suit different types of tasks: the 'straight line' for a simple 'one-off' task; the loop for fixed repetitive tasks; the 'fork' for tasks whose details depend on the particular information received when the program is run.

In designing a program, you will usually find that a combination of these elements is called for: you should choose the type of structure best suited to the particular stage in the task.

4.5.2 Subroutines

BASIC provides one other fundamental way of organising parts of a program, which is suited to tasks that require the same or a very similar sequence of steps in several parts of the program. This structure is provided by the commands GOSUB and RETURN, and is called a 'subroutine'.

Take the following program fragments:

```
80 FOR A=1 TO entries%
90 PRINT name$(A)
100 NEXT
110 PRINT "****Press any key to continue****"
120 WHILE INKEY$="":WEND
130 PRINT:REM Blank line
...
200 FOR A=1 TO queries
210 PRINT equation$(A),result(A)
220 NEXT
230 PRINT "****Press any key to continue****"
240 WHILE INKEY$="":WEND
250 PRINT:REM Blank line
...
```

You will see that lines 110 to 130 and 230 to 250 are identical, apart from their line numbers. This duplication seems a bit wasteful of typing effort and program space, even more so if it occurs more than twice in the program. A more efficient solution is to write the duplicated program lines just once, as a subroutine. A subroutine is simply a sequence of program lines that end with the command RETURN, that can be used ('called') from anywhere in the rest of the program by using the command GOSUB followed by the line number of the first line in the subroutine.

The subroutine should be kept away from the main part of the program to make sure that it will only be entered when the appropriate GOSUB instruction is executed. The best place is usually at the end of the program, after an END, as in the example below.

Using subroutines, this program fragment becomes as follows:

```
...
80 FOR A = 1 TO entries%
90 PRINT name$(A)
100 NEXT
110 GOSUB 1000
...
200 FOR A = 1 TO queries
210 PRINT equation$(A), result(A)
220 NEXT
230 GOSUB 1000
...
990 END
999 REM * subroutine to pause until any key pressed *
1000 PRINT "****Press any key to continue****"
1010 WHILE INKEY$="" :WEND
1020 PRINT:REM Blank line
1030 RETURN
```

GOSUB is rather like the GOTO command, except that when BASIC gets to the RETURN command at the end of the subroutine, it leaves the subroutine and continues execution with the next instruction after the GOSUB instruction that called it originally. If this is not clear, enter then run each of these two programs in turn:

```
10 REM example demonstrating GOTO
20 PRINT "First this, ";
30 GOTO 60
40 PRINT "This instruction never executed"
50 END
60 PRINT "then this, because of the GOTO"
```

Chapter 4: Building larger programs

```
10 REM example demonstrating GOSUB
20 PRINT "First this, ";
30 GOSUB 60
40 PRINT "and finally this, after the subroutine has finished"
50 END
60 PRINT "then this, from the subroutine "
70 RETURN
```

Subroutines can also be used when the program lines are not exact duplicates, by intelligent use of variables and condition testing. For instance, if the messages required in the above example were different, the particular message required could be assigned to a string variable before calling the subroutine, with the subroutine printing that string variable rather than a fixed message.

Subroutines don't just help you write more compact programs. They also make it easier to correct or improve your programs, because you only need to change the subroutine, rather than the many occurrences of the duplicated code throughout the program.

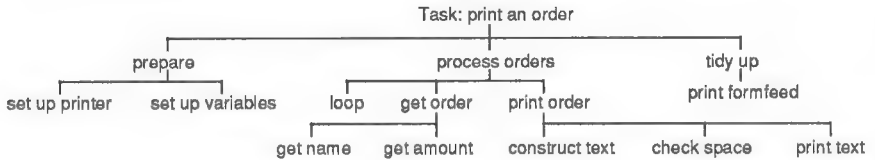
GOSUB may also be used with the ON keyword, to select one from a list of subroutines, depending on the value of a numeric expression. The instruction takes the form:

ON numeric-expression GOSUB line1, line2, line3, etc.

The numeric expression is evaluated when the instruction is executed to yield a number n . If n is between 1 and the number of line numbers in the list, execution continues at the n th subroutine in the list. If n is less than 1 or greater than the number of line numbers in the list, the whole instruction is skipped.

4.5.3 Program design using modules

Subroutines can also be used to write programs that don't have a great deal of duplication, providing a good way of compartmentalising the program. Writing programs this way makes the program much easier to design, develop and understand, by allowing you to break down the task into a hierarchy of subtasks. For example, consider the following task:



Each level in the hierarchy can then be translated into a subroutine. The highest level is very simple:

```

10 GOSUB 1000:REM prepare
20 GOSUB 2000:REM process orders
30 GOSUB 3000:REM tidy up
40 END
50 REM ***** SUBROUTINES *****

```

Each of these major subroutines can now be designed and coded. For example, the main processing subroutine could be:

```

2000 REM *** process orders ***
2010 INPUT "To prepare an order, type Y RETURN",answer$
2020 WHILE answer$="Y"
2030 GOSUB 2100:REM get order
2040 GOSUB 2500:REM print order
2050 INPUT "Another order? type Y RETURN for yes"; answer$
2060 WEND
2070 RETURN

```

Another level down the hierarchy:

```

2100 REM ** get order **
2110 GOSUB 2200:REM get name
2120 GOSUB 2300:REM get amount
2130 RETURN

```

And another:

```

2200 REM * get name *
2210 INPUT "Type the client name followed by RETURN",name$
2220 IF name$="" THEN GOTO 2210
2230 RETURN

2300 REM * get amount *
2310 INPUT "Type the amount, followed by RETURN",amount$
2320 IF amount$="" THEN GOTO 2310
2330 amount=VAL(amount$)
2340 IF amount <= 0 THEN PRINT "The amount must be
                                positive":GOTO 2310
2350 RETURN

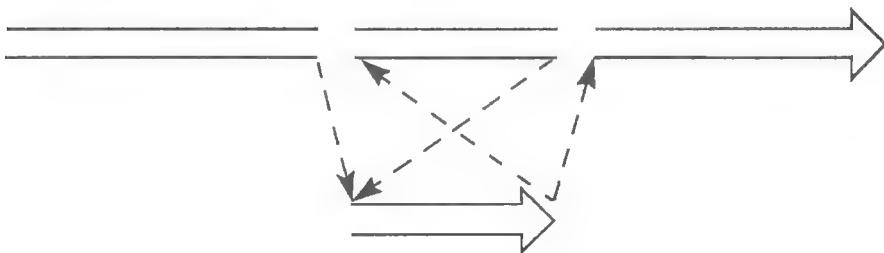
```

And so on ...

By segmenting the program in this way, you can make sure the structure of the program reflects the structure of the task, which in turn makes it easier to find the part of the program that is of interest.

It also allows you to write the program a bit at a time. Starting from the top, you write only one level down at a time, checking that each level works correctly before proceeding to the greater detail of the next level. This makes it easier to spot and correct errors, because you will always know the level which they have developed on.

The flow in a program using a subroutine can be summarised graphically as:



4.5.4 User-defined functions

Subroutines provide a convenient way of 'packaging' instructions. BASIC provides another rather similar facility for packaging expressions, called 'user-defined functions'. These are functions that can be used ('called') anywhere that a BASIC function could be, once they have been defined in the program using a DEF FN instruction.

DEF FN is used in an instruction consisting of four parts:

`DEF FNfunction-name (parameter-list) = function-expression`

The user function call takes the form:

`FNfunction name (list-of-values)`

The function name is chosen by the programmer and can be any valid variable name.

The parameter list (which must be in round brackets) consists of one or more variable names ('formal parameters'), separated by commas. When the function is used, these variables are assigned the values in the corresponding positions in the function call. Note that these variables are entirely separate from any variables of the same name used anywhere else in the program.

The function expression can be any valid expression, but will usually use the values passed to it in the formal parameters to calculate a result. This result is passed back as the result of the user-defined function.

The 'list of values' is a number of values (constants, variables, or expressions), separated by commas. There must be the same number of these values as formal parameters in the corresponding function definition and they must be compatible types ie. both numeric or both string.

Typical functions and function calls (using the example screen control codes) are:

```
10 DEF FNvat (gross)=gross-gross/1.15
20 cost = 100
30 PRINT "The vat on a vat-inclusive item costing ";cost;
      " is ";FNvat (cost)

10 DEF FNmean (n1,n2)=(n1+n2)/2
20 highest%=42:lowest%=3
30 average=FNmean (highest%,lowest%)
40 PRINT average

10 esc$=CHR$(27)
20 REM for a VT52 computer
30 DEF FNscreen$(x,y,text$) = esc$ + "Y" + CHR$(y+32) +
      CHR$(x+32)+text$

40 INPUT "Message ";message$
50 INPUT "Column ";col%
60 INPUT "Row ";row%
70 PRINT FNscreen$(col%,row%,message$)
80 GOTO 40

10 log2 = LOG(2)
20 DEF FNlog2(x) = LOG(x)/log2
30 INPUT "Positive number ";number
40 PRINT "Logarithm to base 2 is ";FNlog2 (number)
50 GOTO 30
```

When it is executed, the user function definition does not do anything other than define the function. It must be executed before the user function can be used. It is usual to group function definitions all in one place, either at the start of the program or in an initialisation subroutine.

User-defined functions provide a convenient but very limited alternative to subroutines when the processing required can be encapsulated into a single expression.

Manipulating information

BASIC functions and operators have been introduced so far in an order and to a depth designed to suit the reader who is entirely new to BASIC.

This chapter describes functions in a little more detail, gathered into logical groups, so you can easily appreciate what facilities are provided in any particular area. It is probably not worth reading about facilities that don't sound relevant to your particular needs, especially if this is your first time working through this user guide; you can always return to them later.

This is not a complete description of the facilities available: the more complex or less useful functions are either not described at all here, or are only described in outline. For further information on any of the functions and operators, see Part II.

5.1 Manipulating numeric information

5.1.1 Arithmetic

The facilities that you will use most frequently with numeric information have already been described: the arithmetic operators of addition (+), subtraction (-), multiplication (*) and division (/).

These operators all apply equally well to integers and floating point (single and double precision) arithmetic. There are two further operators, particularly suited to manipulating integers: \ which is used for integer division (it converts the numbers to integers before dividing and returns an integer result) and MOD, which returns the remainder after performing an integer division:

```
PRINT 7/2
3.5
PRINT 7\2
3
PRINT 7 MOD 2
1
```

5.1.2 Signs

To ensure that a number (expression, variable) is positive, use ABS. This converts a negative or positive number to a positive number of the same size.

To determine the sign of a number (expression, variable), use SGN. This returns 0 if the value is 0, -1 if it is negative, +1 if it is positive. This provides a convenient way of 'compressing' a value's range, for example:

```
100 state$(0) = "overdrawn"
110 state$(1) = "nil stock"
120 state$(2) = "in stock"
...
200 state = SGN(stock(item))+1
210 PRINT name$(item); " is "; state$(state)
etc.
```

5.1.3 Logarithms and exponentiation

There are two logarithmic functions, providing natural logarithms (LOG) and logarithms to base 10 (LOG10).

To convert a natural logarithm back to the corresponding number, use EXP.

To raise a number to a power, use the exponentiation operator, \uparrow . For example, to print 7 to the 4.4th power:

```
PRINT 7  $\uparrow$  4.4
```

To convert a logarithm to base 10 back to the corresponding number, use $10\uparrow x$ (where x is the logarithm).

Note: On many systems, the character you use for the exponentiation operator is ^ and in the rest of this manual, we use ^ rather than \uparrow because it is the character in more general use. When you are using a PCW, simply substitute \uparrow for ^ wherever it appears.

5.1.4 Trigonometry

BASIC provides three trigonometric functions, for converting angles (expressed in radians) into cosines, sines and tangents: COS, SIN and TAN. Each takes a single numeric argument, in brackets. For example:

```
opposite = adjacent*TAN(theta)
```

There is also a single function to convert a tangent back to an angle (in radians):

```
theta = ATN(opposite/adjacent)
```

You can, of course, create further trigonometric functions from these three simple functions. For example, if you need the cotangent of an angle, you can obtain this by putting `1/TAN(theta)` into your expression.

Yet more trigonometric functions can be produced with the help of the other BASIC functions introduced in this chapter. For details, see Appendix IX – or any good mathematics book!

5.1.5 Bitwise operations

A number of operators are provided to manipulate integer numeric information at the level of individual bits. The most useful are:

<code>NOT number</code>	– invert each bit in the number
<code>number1 AND number2</code>	– set bits to 0 unless 1 in both numbers
<code>number1 OR number2</code>	– set bits to 0 unless 1 in either or both numbers
<code>number1 XOR number2</code>	– set bits to 0 unless 1 in one number, 0 in the other

These operators are often used in relational expressions, to invert the sense of a test, for example:

```
WHILE NOT EOF(file%)  
    INPUT #file%,name$  
WEND
```

(Also see EQV and IMP in Part II.)

5.1.6 Random numbers

It is occasionally useful to be able to generate a number whose value you cannot easily predict – a 'random number'. This is often required in statistical analysis, when generating passwords, or programming games. It is very difficult to generate entirely random numbers. The BASIC function RND generates 'pseudorandom' numbers in the range 0...1: these numbers are in an entirely predictable, but rather obscure sequence which will be good enough for most purposes. In general, each time you use RND, you get the next number in the sequence.

So if, for example, you want a random number between 0 and 5, you could obtain this by using:

```
random.number = RND*5
```

(Also see RND and RANDOMIZE in Part II.)

5.1.7 Maxima and minima

If you want to pick out the highest value from a list, use MAX. For example:

```
PRINT "The winner is ";MAX(mark(1),mark(2),mark(3),mark(4))
```

If the values are negative, the highest is the one which is least negative.

Similarly, to pick out the lowest (most negative) number from a list, use MIN.

5.2 Manipulating textual information

5.2.1 Assigning to string variables

To change the whole of a string variable, you will usually use an equals sign:

```
new.string$= "New, new, new"
```

The alternative is to use the special keywords LSET and RSET. These assign one string to another in such a way that the length of the string is preserved by adding extra spaces either at the end of the string (LSET) or at the beginning (RSET). LSET and RSET are chiefly used to assign strings to the 'fields' of the records in a random access file. They are described in Chapter 6, on using disc files.

To change part of a string variable, use MID\$ on the lefthand side of an assignment:

```
date$ = "xxxx March, 1986"  
MID$(date$,1,4) = "21st"  
PRINT date$  
21st March, 1986
```

The parameters to MID\$ specify the string variable, the starting position and the number of characters to be replaced.

5.2.2 Joining strings

The only operator that can be used with strings is the addition operator '+', which adds the string to its right onto the end of the string to its left. Note that this is quite different to adding two numeric values – contrast the result of executing the two following instructions:

```
PRINT "123"+"456"  
PRINT 123+456
```

5.2.3 Splitting strings

BASIC provides three functions to extract part of the text from a string: LEFT\$, MID\$ and RIGHT\$.

- To extract the first n characters from a string, use LEFT\$(text\$, n).
- To extract the last n characters from a string, use RIGHT\$(text\$, n).
- To extract n characters starting from position s in a string, use MID\$(text\$, s , n). (The first character in the string is at position 1.)
- To extract all the characters from position s to the end of the string, use MID\$(text\$, s).

5.2.4 Generating strings

Two functions are provided to generate strings by repeating a single character.

To produce a string of n spaces, use SPACE\$(n).

To produce a string of n characters with internal value c , use STRING\$(n , c).

To produce a string of n characters "z", use STRING\$(n , "z").

For example (where 42 is the internal value of *):

```
PRINT STRING$(20, "*"); "Hi!"; STRING$(20, 42)
*****Hi!*****
```

5.2.5 String length

A string can be anything from 0 to 255 characters long. BASIC provides a function, LEN, that returns the length of the string that is its argument. This is particularly useful when you need to process the string a character at a time, such as in the following routine, which displays the string in title\$ double-spaced across the screen:

```
100 characters = LEN(title$)
110 FOR a = 1 TO characters
120 PRINT MID$(title$, a, 1); " ";
130 NEXT
140 PRINT
```

5.2.6 Searching strings

To check whether a string contains a particular sequence of characters, you can use `INSTR`. This takes the form:

`INSTR ([start,]string1$, string2$)`

Start specifies where the string is to be searched from; if this is omitted or set to 1, the whole string will be searched.

String1\$ is the string to search. *String2\$* is the string to search for in *string1\$*.

`INSTR` returns a numeric result. If the result is 0 then *string2\$* was not found in *string1\$*. Otherwise, the result returned is the position in *string1\$* of the first occurrence of *string2\$* at or after the *start*.

This function has a number of uses not immediately apparent. For example, in the following it is used to cater for the user typing options as upper or lower case letters:

```
...
100 PRINT "A=abort, B=begin again, R=retry. Press a key"
110 answer$ = INKEY$
120 IF answer$ = "" THEN 110
130 code = INSTR(1,"AaBbRr",answer$)
140 IF code = 0 THEN GOTO 100
150 ON code GOSUB 500,500,600,600,900,900
```

5.2.7 Converting strings

Text strings, unlike keywords, will be stored as upper and lower case if that is how they are typed. BASIC provides two functions to convert strings from upper case (capitals) to lower case and vice versa.

To convert a string to all upper case, use `UPPER$(string)`. To convert a string to all lower case, use `LOWER$(string)`. For example:

```
PRINT UPPER$("February 1986 Accounts BOOK")
FEBRUARY 1986 ACCOUNTS BOOK
```

Note that this conversion does not affect any symbols other than the letters 'a'-'z' or 'A'-'Z'.

Characters in text strings can have internal values in the range 0 to 255, but only characters 0–127 are defined by the ASCII standard. You may occasionally need to use strings in which the top bit has been set (for example, as a parity bit) but should now be ignored; use `STRIP$(string)` to ensure that you are only using codes in the range 0 to 127.

5.3 Converting between different types of information

This section describes the functions provided to convert information between the four main forms in which it is stored: string, integer, single and double precision.

Many conversions are performed automatically – for instance, an integer value can simply be assigned to a double precision variable, or a double precision value to an integer variable without using any conversion function. There are two main reasons for using conversion functions: to convert between incompatible data types (numeric and string) and to force a particular conversion when it would not normally occur.

5.3.1 Converting between string and numeric

The individual characters in strings are stored as single bytes, with values in the range 0...255. A different character is displayed for each value. The value corresponding to a particular character is called the character's internal value. For example, the letter 'A' has an internal value of 65, the letter 'B' 66 and so on.

The function `ASC` returns the internal value for a given character, so that `PRINT ASC("A")` would print 65.

The function `CHR$` returns the character with a particular internal value, so that `PRINT CHR$(65)` would print A.

Although numeric values are converted to the text equivalent automatically when printed, you cannot use strings as numbers or vice versa. To use a string (such as "1234") as a number, or a number (such as 1234) as a string, you must use a conversion function.

To convert a number to exactly the same representation as would be used to print it, use `STR$`, for example: `number$ = STR$(number)`

To convert a text representation of a number to a number, use `VAL`, for example: `number = VAL(number$)`

Other functions are provided to convert numeric values to formatted decimal strings, octal and hexadecimal strings. See `DEC$`, `OCT$` and `HEX$` in Part II.

Other functions are also provided to convert numeric values to and from compact string representations for use with disc data files. These are described in Section 6.3.

5.3.2 Converting between integer, single and double precision

When a numeric value of one type is assigned to a numeric variable of a different type, it is automatically converted to suit the new type. However, there are two occasions when you may want to use conversion functions: when the numeric value is not being assigned to a variable so conversion would not normally take place, and when the conversion that would take place is not as you want.

When assigning a floating point value to an integer, the value will normally be rounded to the nearest integer with the integer further away from zero being chosen in the case of a tie. Thus 4.5 would round to 5, while 4.49 would round to 4.

To convert a numeric value to integer, single or double precision (as if assigning it to a variable of that type), use CINT, CSNG or CDBL respectively. This may be needed to force an expression to be evaluated in a particular precision; for example, compare:

Type	a# = 1/10	with	Type	a# = CDBL(1)/10
Type	PRINT a#		Type	PRINT a#
	0.1000000014901161			0.1

In the first case the division is performed using single length arithmetic giving a single length result, which is then converted to double length when it is assigned to the double length variable A#. In the second case the CDBL in the expression converts the dividend to double length before the division, which means that the division is performed using double length arithmetic, giving a double length result. (Incidentally, the first case illustrates the inexact representation of decimal fractions as binary ones, by printing the single length value of 0.1 to sixteen decimal digits rather than the usual seven.)

Functions are also provided to round the floating point part of a number in other ways. These are INT, FIX and ROUND. The number remains floating point.

To round a number to the smaller integer, use INT. INT(3.9) will produce 3, while INT(-3.9) will produce -4.

To round a number by simply losing the decimal part, use FIX. FIX(3.9) will produce 3, while FIX(-3.9) will produce -3.

To round a number to a specified number of decimal places, use ROUND (*number*, *places*). For example, ROUND(3.335, 2) would produce 3.34 while ROUND(-3.335, 2) would produce -3.34. If omitted, *places* is taken as 0, so ROUND(3.335) produces 3.

If *places* is negative, it is taken as the number of figures to round to the left of the decimal place; for example, ROUND(12345, -2) will produce 12300.

Using discs for information storage

The facilities that have been described so far cover most programming needs but are not entirely suited to a major class of tasks that BASIC can be used for: the storage and processing of large amounts of information, such as personnel records, stock lists, address books etc.

So far, discs have only been described as a way of storing programs for later use, but in fact, they can also be used to store information, as a kind of extension of the computer's memory. Information is stored on discs by writing it from variables to named files called 'data files'.

Using data files to store information has a number of advantages and disadvantages compared to using variables:

Files

Information preserved until deliberately erased or file deleted

Limited by disc capacity

Easily shared between programs

Access time varies between nearly instantaneous and several seconds

Information must be transferred to variables before it can be used in instructions

Variables

Information easily lost, eg. when the computer is turned off

Limited by computer memory

Not easily shared between programs

Nearly instantaneous access

Information can be used directly in instructions

To summarise, data files provide a way of preserving information so it can be easily used later with the same or other programs, but take a little more effort and time to use than variables.

BASIC provides three types of data files, to suit different applications: 'sequential', 'random access' and 'keyed access'.

A sequential file is used rather like a DATA statement, containing information in the sequence it was written to the file, which must be read in that sequence.

A random access file is used more like an array, with each element numbered; to read or change a particular element, you simply specify its number.

Keyed access files are rather like random access files, but let you choose the element you want much more conveniently, by using text 'keys' such as a surname or stock description.

Sequential and random access files are described in this chapter; keyed files are described in Chapter 7.

6.1 General disc commands

BASIC provides a number of commands to help you use discs that, although useful when manipulating data files, also have a wider use.

While using the computer's operating system you have access to commands for listing, deleting, renaming and typing files. BASIC provides similar facilities through keywords with the same names and arguments:

- To list files on a disc: *DIR constant*, eg. *DIR *.BAS*
- To delete files on a disc: under CP/M, *ERA constant*, eg. *ERA TEST.**
under MSDOS, *DEL constant*, eg. *DEL TEST.**
- To rename a file: under CP/M, *REN constant=constant*, eg. *REN NEW=OLD*
under MSDOS, *REN constant constant*, eg. *REN OLD NEW*
- To display the contents of a file: *TYPE constant*, eg. *TYPE INFO.DOC*

If you use the MSDOS 2 version of Mallard BASIC, then you can include path-names as well, just as if you were using the operating system itself.

For details of these commands and the significance of their arguments, see guides to the operating system you are using.

The above BASIC commands are followed by constant text without double quotes, just like the operating system commands. BASIC provides equivalent commands that take string expressions as their arguments; the meaning of the argument is the same, but you can use the command just like a normal BASIC command like *LOAD*. These commands are *FILES*, *KILL*, *NAME AS* and *DISPLAY*:

- To list files on a disc: *FILES string*, eg. *FILES file.list\$*
- To delete files on a disc: *KILL string*, eg. *KILL"*.TMP"*
- To rename a file: *NAME string AS string*, eg. *NAME oldname\$ AS newname\$*
- To display the contents of a file: *DISPLAY string*, eg. *DISPLAY name\$+type\$*

BASIC provides another command, RESET, which you should use before changing discs. This tells the operating system that the disc is about to be changed and 'closes' any open files (Section 6.2).

There is also a function, FIND\$, which you can use to check files on a disc. It returns a null string if the file is not there, or a string containing information on the file if found (see Part II). For example, to check if a file is on the disc:

```
IF FIND$(file$) = "" THEN PRINT file$;" not found"
```

6.2 Sequential access files

Sequential access files are the simplest form of data file and so will be dealt with first. Each of the keywords needed to use sequential files will be described first, followed by an example program to illustrate how they work together.

Sequential files can be created and read, but not changed (directly).

6.2.1 Creating a sequential file

Creating a sequential file involves three stages:

- opening the file
- writing information to the file
- closing the file

The file cannot be read until it has been closed.

Opening the file Before you can write to a sequential file, you must tell BASIC about it by opening the file using the keyword OPEN:

```
OPEN "O", #file-number, file-name
```

The "O" is the 'file access mode' and stands for output. This tells BASIC to create an entirely new file, deleting any existing file with the same name. Other modes ("I", "R" and "K") are described elsewhere.

Note that you cannot change parts of an existing sequential file.

The file number is an integer in the range 1–3, which you will use in all subsequent instructions to refer to this file. This number must not be the same as the file number for any other file which is open. (The upper limit of 3 can be changed using the MEMORY command or the /F parameter with the command used to load BASIC – see Part II.) If you need to open more, you will have to close some of the files already open (see CLOSE below).

The file name is the complete name, including file type, that the operating system will use for the new file. This file must not be already open. If it already exists, it will be erased.

The file stays open until you close it (using CLOSE, RUN, BUFFERS, RESET or SYSTEM). You must not remove the disc on which the file has been opened until it has been closed.

Writing information to the file Once the file has been opened for output, you can place information in it using the keywords PRINT # and WRITE #.

Each of these keywords is followed by the file number, a comma, then a list of data items (variables, constants or expressions) which will be written to the file. For example:

```
PRINT #addressfile%,name$(i),address$(i),phone$(i)
```

PRINT # behaves almost exactly as PRINT, allowing the use of USING, SPC and TAB to format the output to the file. The only important difference is that PRINT # does not expand Tab Control codes (Internal value 9) to spaces.

WRITE # is somewhat similar to PRINT #, but writes the data items to the file separated by commas, ignores print zones and writes strings surrounded by double quotes. WRITE # does not allow the use of TAB, SPC or USING keywords.

The difference may be more obvious after the following examples, in which constants have been used for clarity:

```
PRINT "Name","Address";"Telephone";1234
```

would be displayed as:

```
Name           AddressTelephone 1234
```

```
PRINT #file%,"Name","Address";"Telephone";1234
```

would be stored exactly the same, as:

```
Name           AddressTelephone 1234
```

while

```
WRITE #file%,"Name","Address","Telephone";1234
```

would be stored as:

```
"Name","Address","Telephone",1234
```

The need for these two different formats of output will only become apparent when you understand the way that information is read from a sequential file.

You are strongly advised to use `WRITE #` initially, as this automatically produces files which are easy to read correctly using `INPUT #`.

Each time a `WRITE #` or `PRINT #` instruction is executed for the file, the additional information is written to the end of the sequential file, until the file is closed.

(The information is not written to the file immediately, but initially stored in a small area of memory called a 'buffer'. The contents of the buffer are only written to the disc when the buffer becomes full, or when you close the file. This buffering process makes programs more efficient, by reducing the number of times the disc drive has to be accessed.)

Closing the file When you have finished writing to the file, you should close it using the keyword `CLOSE`. This has four main effects:

- ensures that all information written to the file is actually transferred to it from the buffer
- allows the file to be opened for reading
- frees that file number for re-use
- frees resources assigned to that file so another can be opened

To close all files that are open, use `CLOSE` on its own. To close particular files, use `CLOSE` followed by a list of file numbers, separated by commas. For example:

```
CLOSE client.file, stock.file
```

All open files are closed automatically when any of the following commands are executed: `RESET`, `RUN`, `BUFFERS` and `SYSTEM`.

6.2.2 Reading from a sequential file

Reading from a sequential file involves three stages: opening the file, reading the information and closing the file.

Opening the file Before you can read from a sequential file, you must tell BASIC about it, using the keyword `OPEN`:

```
OPEN "I", #file-number, file-name
```

The "I" is the access mode (input). Other modes ("O", "R" and "K") are described elsewhere. The file number is an integer in the range 1–3, which you will use in all subsequent instructions to refer to this file. This must not be the same as the file number for any other file already open. (The upper limit of 3 can be changed, as described above.)

If you need to open more files, you will have to close some of those already open (see CLOSE below).

The file name is the complete name, including file type, that the operating system will use to find the file. You cannot open a file for input if it does not exist or is currently open for output.

The file stays open until you close it (using CLOSE, RESET, RUN, BUFFERS or SYSTEM). You must not remove the disc on which the file has been opened until it has been closed.

Reading information from a sequential file After opening the file for input, you can read information from it using the keyword INPUT #. This is followed by the file number, a comma, and a list of variables to which the information read will be assigned. For example:

```
INPUT #file%,name$(i),address$(i),count(i)
```

Information is read from the file in strictly sequential order (hence 'sequential file'): the first variable in the first INPUT #*n* instruction will be assigned from the first value in the file, the second variable from the second value, and so on. It is not possible to re-read an item; to do so, you must open the file for input again, then read items until you reach the required item again.

The information in a sequential file can be used in two main ways:

- by reading all the data items from the file into suitable arrays, then ignoring the file and using the information in these arrays. This technique can only be used if all the information will fit into the computer memory at the same time, but is generally worthwhile as the particular items of interest can then be reached very quickly, in any order. It is particularly suited to tasks such as 'looking up' information in a file or reorganising information before printing or changing a file.
- by reading data items from the file sequentially, using each item or sequence of related items in turn before reading the next. This technique can be used with files of any size, as only the items just read need to be stored, but limits processing to those 'current' items. It is particularly suited to tasks like totalling figures, copying or printing an entire file.

The type and order of variables in an INPUT # instruction is largely dictated by the structure of the file, which is in turn determined by the instructions used to create it.

The simplest way to use sequential files is to create them with `WRITE #`, then read the information back with `INPUT #`, using the same variable types in the same order when creating and reading the file. For example, if the file was created using:

```
FOR i = 1 TO 10
    WRITE #file%,name$(i),address$(i),telephone$(i)
NEXT
```

then the information could be read back using:

```
FOR entry = 1 TO 10
    INPUT #file%,client$(i),address$(i)
    INPUT #file%,phone$(i)
NEXT
```

Note that you need not use the same names for the variables and the variables need not be grouped exactly the same in the `WRITE #` and `INPUT #` instructions, just be written and read in the same order.

It is not even necessary to use exactly the same variable types as long as they are compatible (ie. both numeric or both string), and you are prepared to accept the rounding that may occur.

(If you want, you can even create files so that numeric data can be read back into numeric OR string variables. This is done by using the `PRINT #` command: refer to Part II for details.)

If you try to read beyond the last data item in a file, this will generate an error. If you know how many items there are in the file when you write the program to read it, you can easily ensure that this limit is not exceeded, as in the example above. Failing this, it is entirely feasible to adopt a convention of always writing a unique value as the last entry in a file, then testing each item read against this.

However, BASIC provides a more elegant solution, in the form of the function `EOF`. `EOF (file%)` returns the value 0 while there are more items to be read from the file and -1 when the last item has been read. For example, to print all the entries in a file named "testfile.seq":

```
file% = 1
OPEN "I",#file%,"testfile.seq"
WHILE NOT EOF(file%)
    INPUT #file%,item$
    PRINT item$
WEND
```

(The keyword `NOT` is used here to invert the result of `EOF (file%)` so that the `WHILE` loop will continue until the end of the file has been reached.)

Closing the file When you have finished reading a file, you should close it using the keyword `CLOSE`. This is not as important as closing a file which has been opened for output, but has three main effects:

- allows the file to be opened for output
- frees that file number for re-use
- frees resources assigned to that file so another can be opened

To close all files that are open, use `CLOSE` on its own. To close particular files, use `CLOSE` followed by a list of file numbers, separated by commas. For example:

```
CLOSE client.file, stock.file
```

All open files are closed automatically when any of the following commands are executed: `RESET`, `RUN`, `BUFFERS` and `SYSTEM`.

6.2.3 Changing a sequential file

BASIC does not provide you with facilities to change a sequential file directly. If you want to change a sequential file (ie. add to, delete or alter information in it), you must do so by creating an entirely new version, from information read from the old version and the changes required.

If the file is small enough, read the entire file into memory, change the information then write the revised version back to the disc, as illustrated in the phone book example below.

Alternatively, you can read information from the file in batches, creating a new version with a different name in stages.

Certain types of change can however be made easily using the random access facilities described in Section 6.3.

6.2.4 Example programs

The following programs illustrate typical uses for sequential files. Don't be put off by their apparent triviality: they have been kept simple deliberately to avoid obscuring the salient points.

Example 1: simple statistical analysis

(This example illustrates the simplest way to use a sequential file – reading it an item at a time, using each data item immediately.)

The task: The task is to total, count and average all the numerical data entries in a sequential file, then print this information together with the maximum and minimum values read, when the whole file has been read. The average should reflect the average magnitude ie. ignore the signs of the numbers, but all other results should be based on the signs.

The data in the file might be financial transactions, number of people at football matches, temperatures; it doesn't matter what. The data items will all be numbers, positive or negative, integer or floating point, in the range -9999.999 to +9999.999. The file may have between zero and twenty thousand entries. The name of the data file will be supplied by the user when the program is run.

If you want to test your understanding of the facilities described so far, attempt to design and write the program now, without reading any further.

The design: Since none of the processing requires access to previously read data, there is no need to try to store the data from the file in an array, which is just as well as larger files would not fit!

The program will consist of the following stages:

- set up variables and screen
- choose disc
- choose file
- read and process entries
- finish calculations and print results
- tidy up

Chapter 6: Using discs for information storage

The Program: The following is one way of coding the functions required by the design.

```
10 REM example 1 - simple statistical analysis
20 REM *** set up variables ***
30 '
40 file%=1: minimum = 10000: maximum = -10000
50 '
60 PRINT "Statistical analysis program"
70 '
80 REM *** choose disc ***
90 '
100 FILES
110 PRINT "If the file you want to analyse is not on this disc,"
120 PRINT "change discs then press C."
130 PRINT "If this is the right disc, press any other key
                                     to continue"
140 i$ = ""
150 WHILE i$ = ""
160 i$ = INKEY$
170 WEND
180 i$ = UPPER$(i$): IF i$ = "C" THEN RESET: GO TO 100
190 '
200 REM *** choose file ***
210 '
220 INPUT "Type the data file name, followed by RETURN";filename$
230 IF FIND$(filename$) = "" THEN PRINT filename$; " IS
    NOT ON THE CURRENT DISC - TRY AGAIN": GOTO 220
240 '
250 REM *** read and process entries ***
260 '
270 PRINT "Analysing ";filename$; " - PLEASE WAIT": PRINT
280 OPEN "I",#file%,filename$
290 WHILE NOT(EOF(file%))
300   count% = count% + 1
310   INPUT #file%,number
320   total = total + number
330   total.absolute = total.absolute + ABS(number)
340   minimum = MIN(number,minimum)
350   maximum = MAX(number,maximum)
360 WEND
```

```
370 '  
380 REM *** finish calculations and print results ***  
390 '  
400 IF count% = 0 THEN PRINT filename$;" is empty":GOTO 500  
410 average = total.absolute/count%  
420 PRINT "RESULTS FOR FILE ";filename$:PRINT  
430 PRINT "Final total: ";TAB(40);total  
440 PRINT "Average absolute value of an entry: ";TAB(40);average  
450 PRINT "Minimum value was: ";TAB(40);minimum  
460 PRINT "Maximum value was: ";TAB(40);maximum  
470 '  
480 REM *** tidy up ***  
490 '  
500 CLOSE #file%  
510 END
```

You should find that this program is largely self-explanatory. The only part that is slightly complex is lines 140–170. These use the `INKEY$` function to check the keyboard over and over again until a key has been pressed, and then pass that key, stored in `i$`, to line 180.

While this program is a reasonable solution, it is by no means perfect. For example, the user has to press Control-C to quit, the variables should perhaps be double precision, the results could be formatted more neatly. Try enhancing the program along these lines.

Testing: The final stage in developing the program should be to test it, in as realistic a manner as possible. As you haven't got any of the files that the program is designed to work with, you must generate them. In keeping with the simple task, a simple program to generate a simple test file:

```
1  REM test1  
10 OPEN "O",1,"test1.seq"  
20 FOR a = 1 TO 10  
30   WRITE #1,a,-a  
40 NEXT  
50 CLOSE 1
```

Run this program to generate the test file, then analyse it using the analysis program.

The ideal test produces results that you can easily check by independent means while pushing the program to its designed limits. A single test will rarely satisfy both aims. This simple test produces results that you can easily check by doing a little arithmetic, but does not include the number nor the wide range of numeric values that the specification requires the program to be able to cope with.

To save you the trouble of the arithmetic, the results should be:

Final total:	0
Average absolute value of an entry:	5.5
Minimum value was:	-10
Maximum value was:	10

Example 2: phone book

(This example shows how to use a sequential file to set up an array for processing in memory, reading and writing the file when needed.)

The task: The task is to provide a 'phone book', which will give the phone number corresponding to a name typed by the user, and allow the user to add names and phone numbers to the 'book'. To keep the program simple, other useful facilities such as editing existing entries and printing a sorted phone list will not be provided.

When specifying a name to search for, upper and lower case are to be considered equivalent, the whole name need not be specified and all entries matching the name will be displayed.

The maximum number of entries will be 100. The longest phone number (including the STD code) will be 12 digits. The STD code can be stored as integral part of the number. The longest name will be 30 characters.

The phone book file will have a fixed name.

If you want to test your understanding of the facilities described so far, attempt to design and write the program now, without reading any further.

The design: As the number of entries is fairly small, the whole file can be read into an array in memory for ease and speed of processing. For the same reason, there is no need to sort the entries into a particular order; the computer can find any particular name quickly by checking each element in turn.

Part of the design is deciding the way that the information will be stored. There are three types of information in the file: the names and phone numbers (obvious) and **the relationship between a name and a phone number**. This could be stored in a number of ways:

- as a list of names, list of phone numbers (the third name has the third number)
- as a list of entries consisting of a name followed by its phone number
- as a list of entries consisting of a name plus a list of indices to the phone numbers in a subsequent list (eg. "AdAstra Inc",4,7 means that AdAstra Inc has phone numbers 4 and 7 in the phone number list)

There is not much to choose between the first two methods. The third, while more complex, can cater better for more complex situations (such as one name having several phone numbers, or several names having a shared phone number).

The program will consist of the following stages:

- set up variables and screen
- read file and store entries
- ask for and carry out instructions (search by name, add an entry, finish)
- write file and tidy up

The program: The following is one way of coding the functions required by the design. Note in particular that the program cannot **change** the phone book file as such; it must change the information held in memory, then write the file all over again!

```

10 REM example 2 - phone book
20 '
30 REM **** set up variables ****
40 DIM name$(100),phone$(100)
50 phone.file$ = "phone.seq": file% = 1
60 false = 0: true = -1
70 file.changed = false
80 PRINT "Phone book"
90 '
100 REM **** get data ****
110 GOSUB 290
120 '
130 REM **** get commands ****
140 command = 0
150 WHILE command < 5
160 PRINT "Phone book ";entries;" entries"
170 PRINT : PRINT "Search for number, Add a number or
                                Finish(S/A/F)"
180 match$ = "SsAaFf": GOSUB 850: command = answer
190 ON command GOSUB 460,460,580,580:GOSUB 940
200 WEND
210 '

```

Chapter 6: Using discs for information storage

```
220 REM **** finish ****
230 IF file.changed = true THEN GOSUB 710
240 END
250 '
260 REM * * * * * SUBROUTINES * * * * *
270 REM *** read file ***
280 REM ** check if there first! **
290 IF FIND$(phone.file$) <> "" THEN GOTO 370
300 PRINT "No phone book file on this disc."
310 PRINT "Change disc or start New phone book (C/N)?"
320 match$ = "CcNn": GOSUB 850
330 IF answer = 3 OR answer = 4 THEN entries = 0: RETURN
340 GOSUB 800: GOTO 290
350 '
360 REM ** read entries **
370 PRINT: PRINT "Reading phone book: please wait"
380 OPEN "I", #file%, phone.file$
390 entries = 0
400 WHILE NOT (EOF(file%))
410   entries = entries + 1
420   INPUT #file%, name$(entries), phone$(entries)
430 WEND
440 CLOSE file%
450 RETURN
460 REM *** search ***
470 IF entries = 0 THEN PRINT "Phone book empty!!!": RETURN
480 PRINT "Type name to search for followed by RETURN,"
490 INPUT "or just RETURN to skip search", search$
500 IF search$ = "" THEN RETURN
510 count = 1
520 WHILE count <= entries AND name$(count) <> search$
530   count = count + 1
540 WEND
550 IF count > entries THEN PRINT search$;" not
                                                                    found": RETURN
560 PRINT "Phone number: "; phone$(count): RETURN
570 '
580 REM *** add entry ***
590 IF entries >= 100 THEN PRINT "No room!!!": RETURN
600 PRINT "Type name to add followed by RETURN,"
610 INPUT "or just RETURN to skip entry"; name$
620 IF name$ = "" THEN RETURN
630 PRINT "Type phone number to add followed by RETURN,"
640 INPUT "or just RETURN to skip entry"; phone$
650 IF phone$ = "" THEN RETURN
```

```
660 entries = entries + 1
670 name$(entries) = name$
680 phone$(entries) = phone$
690 file.changed = true: RETURN
700 '
710 REM *** write file ***
720 PRINT: PRINT "Writing phone book: please wait"
730 OPEN "0", #file%, phone.file$
740 FOR count = 1 TO entries
750   WRITE #file%,name$(count),phone$(count)
760 NEXT
770 CLOSE #file%
780 RETURN
790 '
800 REM * change disc *
810 RESET
820 INPUT "Insert disc with phone book on and press RETURN",a$
830 RETURN
840 '
850 REM * get key *
860 answer$ = ""
870 WHILE answer$ = ""
880 answer$ = INKEY$
890 WEND
900 answer = INSTR(match$,answer$)
910 IF answer = 0 THEN GOTO 860
920 RETURN
930 '
940 REM * wait for key *
950 PRINT: PRINT "Press any key to continue"
960 WHILE INKEY$ = ""
970 WEND
980 RETURN
```

Again, this program is by no means perfect. For example it will not find a name if you type it in capitals when the phone book has it in small letters, or if you do not type the complete name. Similarly, when you add an entry, it does not check if there is already an entry with the same name, it simply places it at the end of the file.

Try changing the program to correct these deficiencies. (HINTS: Use `UPPER$` or `LOWER$` to convert the entry being compared and the search string to the same case. Use `LEN` and `LEFT$` to allow incomplete names to be found. Search through the array each time when adding and replace an existing entry that matches, only adding new ones to the end, but be careful to adjust the 'entries' count accordingly.)

Testing: To test the program, run it to create a new phone book and add a few entries. Run it again, to check that it finds the existing phone book. Try all three commands to see if they work. If you've got a lot of patience, check that it will handle the limit of 100 entries correctly!

6.3 Random access files

Random access files are quite similar to sequential files: they also have names, must be opened before they are used, can have information written to or read from them and should be closed after use.

Random access files have two major advantages: the information in them can be read in any order (hence random rather than sequential access) and that information can be changed, also in any order. (Information in sequential files can only be read in the order that it was written and cannot be changed directly – you must read the file into memory (in one go, or in chunks), changing any information that must be altered while it is in memory, then write it back to a new file on the disc.)

These differences make random access files more convenient and easier to use than sequential files for many applications. Against these advantages, random access files are a little more difficult to understand and use properly, and tend to take up more space on the disc for the same amount of information.

Information is stored in a random access file rather differently, too. In a sequential file, each data item is written or read separately and the different items can have different lengths. The items read from or written to a random access file are called 'records'. All the records in a file have the same fixed length. Each record can consist of a single data item (just like a sequential file) or a number of data items that are handled together.

Records provide a neat way of storing related pieces of data. For example, in a personnel file, there could be a record per person, containing the following data items:

- Name (30 characters)
- Address (50 characters)
- Phone (10 characters)
- NI number (9 characters)
- Date of birth (6 characters)
- Pay scale (1 character)
- Insurance scale (1 character)

Keeping all this related information together makes a lot of sense!

To indicate the particular record that you want to read or write, you use its position in the file – its 'record number'. The first record is number 1, the second number 2, and so on. This is rather like the way you use array variables, by specifying the index (eg. `name$(2)`). In fact, you can use random access files rather like arrays of variables, which are no longer limited in size by the computer memory and can be shared easily between programs!

Random access files are often used to hold information on individual items, such as people, companies or products with one or a fixed number of records being used per item. Programs can then be designed to allow the users quickly and easily to select record(s) to view, change, print etc. The best way to design a file for this type of application is to position records in the file according to some numeric information that users associate with the item, such as an employee's code or an item's stock number. If this is not possible, you will usually have to write down the record number for each item, and have users specify the information required by this number, so that 'ACME Motor Insurance' becomes company 1, 'ACME National' company 2, etc. (Keyed files provides a much more elegant solution however – see Chapter 7!)

6.3.1 Creating a random access file

A random access file is created in five separate stages:

- opening the file
- defining the record layouts
- assigning data to the record
- writing the record to the file
- closing the file

Each of these will now be described in turn.

Opening the file Before you can write information to a random access file, you must open it using the keyword **OPEN**:

```
OPEN "R", #file-number, file-name
```

For example:

```
OPEN "R", #3, "person2.fil"
```

The "R" indicates that the file is open for Random access (rather than "O" – sequential Output, or "I" – sequential Input, or "K" – Keyed access).

The significance of the file number and file name is as described for opening a sequential file, above.

Chapter 6: Using discs for information storage

You can also include another parameter at the end of the instruction to specify the record size required, if the usual size of 128 characters is not acceptable, ie:

`OPEN "R",file-number,file-name,record-size`

This allows you to use disc space more efficiently when the data in each record is significantly less than 128 characters. It also allows you to use larger record sizes when 128 characters is not enough, but you must first change the maximum buffer size using MEMORY – see Part II.

Opening a file for random access creates it, if it does not exist already, like opening it for sequential output. However, if the file already exists, it is not deleted by opening it for random access. This allows you to add or change information in a random access file, as described below. It does mean that if you want to be sure of creating an entirely new random access file, you should check if there is a file with the same name on the disc first and delete it if found (using FIND\$ and KILL as described above).

Opening a random access file makes BASIC reserve a space in memory for it, called a 'record buffer'. This is the same size as the record length (usually 128 characters). It is here that BASIC will assemble the information that you want to write for each record. Each file opened for random access has its own record buffer.

Although it is possible to open a random access file which is already open, this is not recommended.

Defining the record layouts Information is placed in the record buffer (to prepare the record for writing to the file) by assigning it to special variables, called 'field variables'. These are a special form of string variable, set up using a FIELD instruction. This defines the record layout by dividing the record buffer up into individual areas(fields), each of which is identified by a field variable.

The FIELD instruction takes the form:

`FIELD #file-number,field-size AS field-variable[,field-size AS field-variable]`

For example:

`FIELD #3, 10 AS name$, 30 AS address$, 10 AS phone$`

The file number is the number that you used to open the random access file; note that the record layout is only usable with this one file; it is not automatically available for all files.

The field size specifies the number of characters that the field variable will have reserved for it in the record. Field variables must be string variables. The total number of characters (ie. the sum of all the field sizes in the instruction) must not exceed the record length (usually 128, but see OPEN in Part II).

Note that the FIELD instruction does not write any information to the file, it simply labels parts (fields) of the record with the field variable names, to make it easy for you to set up records for writing to the file (see below). Any previous information stored in these field variables is lost; they immediately take the current contents of their field in the record.

You can define as many record layouts as you like for the file, so that you can write records with quite different layouts to the same file. For example, in the personnel file described above, you might occasionally want to use a different record type – a 'continuation record' – for an employee with a complex address. You would then need to define two different record layouts.

The main record type:

```
FIELD #perfile%, 30 AS name$, 50 AS address$, 10 AS  
        phone$, 9 AS ni.number$, 6 AS birth.date$, 1 AS  
        pay.scale$, 1 AS ins.scale$, 1 AS cont.record$
```

The 'continuation record' type:

```
FIELD #perfile%, 108 AS address.cont$
```

Field variables are rather special because of their fixed length (as defined in the FIELD instruction) and should only ever be assigned as described below.

Assigning information to a record Information is placed in a record by assigning it to one of the string variables that have been associated with the file by a previous FIELD instruction.

This assignment must always take one of the following three forms:

LSET field-variable = string-expression

RSET field-variable = string-expression

MID\$ (field-variable, start, length) = string-expression

Assigning to a field variable using any other method breaks its association with the file, and uses it as a normal string variable again. The value assigned to the variable is not placed in the record buffer.

The LSET command assigns characters from the string expression to the field variable, left justified. If the string expression is shorter than the field variable, spaces will be added to its (righthand) end to fill the field. If the string expression is longer than the field variable, the surplus characters (at the the righthand end) will be discarded without error.

The RSET command behaves similarly, except that spaces will be inserted at the lefthand end if the string expression is too short.

The MID\$ command assigns *length* characters from the lefthand end of the string expression to the field variable, starting from *start*.

For example:

```
FIELD file%,20 AS name$
```

After:

```
LSET name$ = "*****"  
name$ is: "*****"
```

After:

```
LSET name$ = "Anne Elizabeth"  
name$ is: "Anne Elizabeth"
```

After:

```
RSET name$ = "Catherine Louise"  
name$ is: "Catherine Louise"
```

And then after:

```
MID$(name$,2,3) = "*****"  
name$ is: "***Catherine Louise"
```

Numeric information can be assigned to a field by first converting it to the equivalent string:

```
LSET number$ = STR$(count)
```

However, this can be a very inefficient way to store numbers; for example, an integer such as -30000 that BASIC stores in two bytes would take up 6 characters in a file using this method. Also, some accuracy may be lost when converting from the internal binary representation to decimal. BASIC solves both potential problems by providing three functions for converting numeric information to a more compact 'string' form: MKD\$, MKI\$ and MKS\$:

- To convert a double precision number to an eight-character string, use MKD\$(*number*).
- To convert a single precision number to a four-character string, use MKS\$(*number*).
- To convert an integer to a two-character string, use MKI\$(*number*).

For example:

```
LSET number$ = MKI$(total)
```

These 'strings' will not display or print correctly.

Complementary functions are provided to convert these strings back to the numeric form (CVD, CVI and CVS) and are described in more detail below.

(Information can also be assigned to the record using PRINT #*n* or WRITE #*n*, but that will not be described further here; see Part II for details.)

Writing the record Having opened a random access file, defined a record layout for it and assigned data to that record, you are now ready to write the record to the file.

Records are written using the keyword PUT, which takes the form:

PUT *#file-number*

OR

PUT *#file-number, record-number*

The file number is the file number you used originally when opening the file.

Use the first form to write the file rather like a sequential file; each record is written after the last one written using PUT (or read using GET – see below). This is the quickest way to write a random access file, as the disc drive does not have to keep finding different parts of the file.

The second form will mainly be used when changing an existing file or creating a file with a complex structure. The record number is the position that you want the record to have in the file. Note that the length of the file is dictated by the highest record number written to it and the record length (usually 128 characters); if you write a single record using PUT *#file%, 1000* then there will be 1000 records in the file – 999 unused and one used!

Writing a record to the file does not change the record buffer; this is only changed by assigning to the field variables or reading the file using the same file number.

Closing the file A random access file should be closed when you have finished writing it, just like a sequential file, to ensure that all the information has been written to the disc.

To close all files that are open, use CLOSE on its own. To close particular files, use CLOSE followed by a list of file numbers, separated by commas. For example:

```
CLOSE #client.fil, #stock.fil
```

6.3.2 Reading a random access file

A random access file is read in five separate stages:

- opening the file
- defining the record layouts
- reading records from the file
- using the information from the record
- closing the file

Opening the file Before you can read information from a random access file, you must open it using the keyword OPEN:

```
OPEN "R", #file-number, file-name
```

For example:

```
OPEN "R", #3, "person2.fil"
```

The "R" indicates that the file is open for Random access (rather than "O" – sequential Output, or "I" – sequential Input, or "K" – Keyed access).

The significance of the file number and file name is as described for opening a sequential file, above.

You can also include another parameter at the end of the instruction to specify the record size required, if the usual size of 128 characters is not acceptable, ie:

```
OPEN "R", #file-number, file-name, record-size
```

This will usually be used when reading a file that was created with a record size of other than 128 characters, as it is usual (although not essential) to use the same record sizes when reading and when writing a file.

Defining the record layouts The simplest way to read a file is to define the record length and record layouts to be the same as those used to write it. In this way, you can be sure to read the information back the way it was written. There are good reasons for reading a file with a different record length, or with a different record layout, but they are beyond the scope of this introduction.

Record layouts are defined using FIELD, exactly as described for creating a file, above.

Reading records from the file To read data from a random access file, you must first read the record containing it, using the command GET:

```
GET #file-number
```

or

```
GET #file-number, record-number
```

Using GET without a record number reads the next record after the last one read (using GET) or written (using PUT) using this file number. If the file has only just been opened, it reads the first record (record number 1).

Using GET in this way you can treat a random access file a little like a sequential file.

Using GET with a record number you can read any record in the file.

Detecting the end of a random access file is a little more difficult than for a sequential file. You can still use the EOF function, but must read the file at least once before doing so. Also, EOF may return a 'true' value even when you have not attempted to read beyond the end of the file, if the record you have just read was never written to (ie. is an empty record).

In practice, this means that if a program needs to be able to find the end of a random access file, that file must either be of a fixed length (which the program knows), or each record in the file must have been written to when it is created (using a suitable null value for empty records, such as spaces or binary zeros).

Using information from the current record Information from the last record read for a particular file number is used by using the field variables as normal string variables. Note that the contents of these variables will change as soon as another record is read using this file number.

If there is more than one record layout defined for this file number, you can use the field variables from the different layouts in any combination.

Numeric information written to the file in the compact string form by using the MKD\$, MKI\$ and MKS\$ functions should be converted back to the equivalent numbers using the complementary functions CVD, CVI and CVS:

- To convert a double precision string to double precision number, use CVD(*string*).
- To convert a single precision string to a single precision number, use CVS(*string*).
- To convert an integer string to an integer, use CVI(*string*).

(Information can also be read from the record using INPUT #, but this will not be described further here.)

Closing the file When a program has finished reading a random access file, it should close it using the keyword CLOSE.

To close all files that are open, use CLOSE on its own. To close particular files, use CLOSE followed by a list of file numbers, separated by commas. For example:

```
CLOSE #personnel%, #paye%
```

Although not as essential as when the file has been written or changed by the program, closing the file releases its record buffer and other system resources for use with other files.

6.3.3 Reading, writing and changing a random access file

As remarked above, the instructions used to open a random access file and define record layouts are exactly the same, whether the file is going to be read from or written to. In fact, you can easily read, change, then rewrite records when working with a random access file, by simply combining the keywords introduced above. This is illustrated by the following example program.

6.3.4 Examples

Example 3 – simple personnel file

The task: To create, maintain and interrogate a simple personnel file. The file to be able to handle up to 500 personnel. The information to be stored is:

- Employee number (3 digits)
- Name (30 characters)
- Address (50 characters)
- Phone (10 characters)
- NI number (9 characters)
- Date of birth (6 characters)
- Pay scale (1 character)
- Insurance scale (1 character)

The facilities required are:

- add a record
- delete a record
- replace a record
- display a record

Records to be selected by employee number.

If you want to test your understanding of the facilities described so far, attempt to design and write the program now, without reading any further.

The design: The maximum amount of information to be handled (approximately 50,000 characters) precludes storing the data in a sequential file and processing it in memory, as does the need to be able to update the records easily.

Using a random access file to store the data, the record format must be decided next. Since records will be requested by employee number, the obvious order in which to put records is by that number. Since the employee number is 3 digits, this would produce a file of 999 records, which would be half full if the design limit of 500 employees is reached. This seems acceptable, so there is no need for more complex code: the record number for an employee will be the employee number. Note that this means that the employee number does not need to be stored in the record as it is given by the record number!

The overall design, then, is:

- set up variables and screen
- open file and set up record layouts
- ask for and carry out instructions
- close file and tidy up

The program: The example program below has been kept deliberately simple so as not to obscure the use of the random access keywords.

```
10 REM example 3 - personnel file program
20 '
30 REM ****set up variables****
40 file% = 1: limit = 500
50 '
60 REM ****user-defined functions****
70 '
80 DEF FNhead$(title$) = STRING$( (74-LEN(title$))/2,"") +
      " " + title$ + " " + STRING$( (76-LEN(title$))/2,"")
90 '
100 REM ****set up screen****
110 '
120 PRINT FNhead$("Personnel files")
130 '
140 REM ****set up file****
150 '
160 IF FIND$("person.rnd") = "" THEN GOSUB 1100
      ELSE OPEN "R", #file%, "person.rnd"
170 FIELD file%, 30 AS rec.name$, 50 AS rec.address$, 10 AS
      rec.phone$, 9 AS rec.ni$, 6 AS rec.birth$, 1 AS rec.pay$,
      1 AS rec.ins$
180 FIELD file%,128 AS record$
190 '

```

Chapter 6: Using discs for information storage

```
200 REM ****get a record****
210 '
220 PRINT: PRINT
230 INPUT "Type personnel number (1 to 500),
                                then RETURN"; person
240 IF person <1 OR person >limit THEN GOTO 230
250 '
260 PRINT FNhead$("Personnel number"+STR$(person))
270 PRINT
280 GET file%,person
290 '
300 REM ****get and obey an instruction****
310 '
320 IF rec.name$ = STRING$(30," ") THEN GOSUB 430
                                ELSE GOSUB 520: PRINT: PRINT
330 prompt$ = "Press F to finish or C to continue"
340 match$ = "FfCc":GOSUB 1220
350 IF answer >2 THEN GOTO 220
360 '
370 REM ****tidy up****
380 '
390 CLOSE
400 PRINT cls$
410 END
420 '
430 REM ****subroutines****
440 '
450 REM ***empty record options***
460 '
470 PRINT "This employee not known"
480 match$ = "YyNn":prompt$ = "Add new record (Y/N)":GOSUB 1220
490 IF answer = 1 OR answer = 2 THEN GOSUB 800
500 RETURN
510 '
520 REM ***existing record options***
530 '
540 REM **print record**
550 '
560 PRINT "Name: ";rec.name$
570 PRINT "Address: ";rec.address$
580 PRINT "Phone number: ";rec.phone$;TAB(40);"Date of
    birth: ";LEFT$(rec.birth$,2);"/"; MID$(rec.birth$,3,2);
                                "/";RIGHT$(rec.birth$,2)
590 PRINT
600 PRINT "Nat. Ins No.: ";rec.ni$;TAB(30);"Pay scale: ";
                                rec.pay$;TAB(60); "Ins. scale: ";rec.ins$
```

```
620 REM **offer and obey options**
630 match$ = "CcDdPpSs"
640 prompt$ = "Change, Delete, Print or Skip (C/D/P/S)"
650 GOSUB 1220
660 ON answer GOSUB 800,800,690,690,1320,1320
670 RETURN
680 '
690 REM ***delete current record***
710 match$ = "YyNn"
720 prompt$ = "About to delete record. Type Y to delete, N
                                                    not to"
730 GOSUB 1220
740 IF answer >2 THEN RETURN
750 LSET record$ = ""
760 PUT #file%,person
770 PRINT "Record deleted"
780 RETURN
790 '
800 REM ***add/change record***
820 INPUT "Name";name$
830 INPUT "Address";address$
840 INPUT "Phone number";phone$
850 INPUT "Date of birth: day in month";day$
860 INPUT "month in year";month$
870 INPUT "year (last 2 digits)";year$
880 INPUT "Nat. Ins No";ni$
890 INPUT "Pay scale";pay$
900 INPUT "Ins. scale";ins$
910 '
920 match$ = "YyNn"
930 prompt$ = "OK? Press N to retype, Y to continue"
940 GOSUB 1220
950 IF answer > 2 THEN 820
960 '
970 LSET rec.name$ = name$
980 LSET rec.address$ = address$
990 LSET rec.ni$ = ni$
1000 LSET rec.pay$ = pay$
1010 LSET rec.ins$ = ins$
1020 MID$(rec.birth$,1,2) = RIGHT$("0"+day$,2)
1030 MID$(rec.birth$,3,2) = RIGHT$("0"+month$,2)
1040 MID$(rec.birth$,5,2) = RIGHT$("0"+year$,2)
1050 LSET rec.phone$ = phone$
```

Chapter 6: Using discs for information storage

```
1060 PUT #file%,person
1070 PRINT "New/changed details written"
1080 RETURN
1100 REM ***create empty file***
1111 '
1120 PRINT "Creating empty file - please wait"
1130 OPEN "R",#file%, "person .rnd"
1140 FIELD file%,128 AS record$
1150 LSET record$ = ""
1160 FOR a = 1 TO 500
1170   PUT #file%
1180 NEXT
1190 PRINT "File created"
1200 RETURN
1210 :
1220 REM **get key**
1240 PRINT: PRINT prompt$
1250 answer$=""
1260 WHILE answer$ = ""
1270   answer$= INKEY$
1280 WEND
1290 answer = INSTR(match$,answer$) : IF answer = 0
                                THEN GOTO 1250
1300 RETURN
1310 '
1320 REM **print record**
1340 LPRINT "Personnel record for employee number";person
1350 LPRINT "*****"
1360 LPRINT "Name: ";rec.name$
1370 LPRINT "Address: ";rec.address$
1380 LPRINT "Phone no: ";rec.phone$;TAB(40);"Date of
    birth: ";LEFT$(rec.birth$,2);"/";MID$(rec.birth$,3,2);
                                "/";RIGHT$(rec.birth$,2)
1390 LPRINT
1400 LPRINT "Nat Ins No: ";rec.ni$;TAB(30);"Pay scale: ";
    rec.pay$; TAB(60);"Ins. scale: ";rec.ins$
1410 RETURN
```

Many desirable facilities have been omitted, such as checking the validity of dates typed, being able to print lists of personnel records or find personnel by information other than the employee name. You may care to design and implement these enhancements yourself.

Testing: Devise tests to check the basic functions of the program; that it creates a personnel file when there isn't one on the disc, doesn't when there is, and that the record display, add and delete facilities work correctly.

Keyed access files for data bases

The Random access files described in the previous chapter let you pick out information from the file in any order, so they could well be used to store data bases such as address lists, personnel files etc. However, Mallard BASIC offers a very much better way of storing data bases – Keyed access files.

The problem with using Random access files for data bases is that they work in the same way as arrays. To pick out a particular piece of information, you need to know its number in the file – just as you have to know which element of an array contains the value you want. This method of selection is fine when the items of information are arranged in a nice tidy order or there are few enough items of information that you can search through until you find the information you want, but it is not very practical when you have the amount of data you typically find in a data base. Then, your only hope of finding a piece of information is by keeping an index.

Creating and maintaining an index is not a simple proposition. You would have to set up a separate file to hold the index and then arrange for your program to search through the current index, sort new entries into alphabetical order, merge them with the existing index, and store the new version of the index each time you added anything to the data base.

With Keyed access files, an index is automatically created and maintained for you. Moreover, these files use a very sophisticated type of index called a B*-tree which reduces the effort your computer has to make to find a record. It also ensures that your records are sorted into alphabetical order of keyname: you don't need to have a special (complicated) sorting program to do this for you.

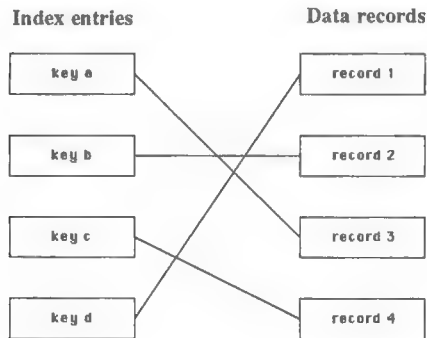
To pick out pieces of information from a Keyed access file, you simply quote one or more of the 'Keys' that have been associated with the information you want. For example, if the data file is an address list, you might have chosen to tag Fred Smith's address in Yorkshire with the keys 'Smith' and 'Yorkshire' – so that you can pick out the full information by telling BASIC to look for Smith or Yorkshire, or both Smith and Yorkshire as appropriate.

Each Keyed file is actually two files: a Data file and an Index file. The Data file is essentially a Random access file, holding all your information. The Index file is a

Chapter 7: Keyed access files for data bases

special file that holds all the keys in up to eight different indexes, known as Ranks. Mallard BASIC lets you deal with the Data file and the Index file as though they were one file.

We can represent the Data and Index files as follows:



Each entry in the Index file just consists of the key value itself plus a pointer to the appropriate record in the Data file – the record number. An individual index entry can only point to one data record, but a number of index entries can point to the same data record.

Programs involving Keyed files use standard BASIC commands to control how the program works, but special 'Jetsam' commands to read and write the Keyed files. In fact, many of these commands are special versions of standard BASIC file handling commands – OPEN, GET, PUT, CLOSE, etc. The extra commands that are involved are ones to flip through the index to see what's there.

The actual order of the records in the Data file never affects how you use a Keyed file because you never search through the data file itself; all your searching is through the index. It is the order of the keys in the Index file and the fact that the keys are in alphabetical order that are important.

The last key that was looked up in the Index file and the record that this pointed to have an important role in Keyed file handling. Together they define the Current position in the Keyed file. When you design a program that uses a Keyed file, it helps to be aware of how the Current position will change as you work through the program. As a rule, changes to the Keyed file are made at the Current position and so a number of the special Jetsam commands either need you to set the Current position first or change it themselves.

The commands that work with Keyed files all require you to set a 'lock'. This tells BASIC whether you want exclusive use of the file (or of a particular record) or

whether other users may read the file while you are working on it. The lock is only needed for multi-user systems where you could have a number of people wanting to use the same file at the same time. It is included in all versions of these commands so that programs can readily be transferred between single-user and multi-user systems. In this chapter, we will be simply specifying exclusive access to both files and records because a program that uses exclusive locks throughout will work in the same way on a multi-user system as it does on a single-user one. How to use locks on multi-user systems is described in Part II, Section 9.2.

Writing a program that uses Keyed files

Any program in which Keyed files are used has three main stages:

- an initial preparatory stage, in which an area of memory is allocated for manipulating the Keyed file, the Keyed file is opened and the record layout is defined
- the main part of the program in which the Keyed file is searched and updated
- a final stage, in which the Keyed file is closed.

This final closing stage might seem a refinement but it is in fact very important. The way Keyed files are handled means that there are times when the Index and Data files stored on disc are slightly out of step. If you were to stop at that point, your data base would no longer be correct. When the file is closed correctly, both files are brought fully up to date. Rather than let you pull the wrong information out of the data base (probably unnoticed), BASIC won't let you use a Keyed file that wasn't correctly closed the last time it was used – so be careful!

To illustrate how to write these different parts of the program, we shall now write a simple data base program that reads an address list keyed by name, adds new entries and deletes entries that are no longer required. So that the Keyed file handling isn't obscured, the program won't worry about vetting the input or any of the other refinements that you might build into a 'real' program.

Don't worry if you don't understand all the elements of this program. You will still get a good idea of how to go about the task and the example can be used as a model for when you want to write your own data base program.

Note: Data base programs as a rule work with existing files and so you need to use a separate program to set up the Keyed file in the first place. This program just has to Create the Keyed file, defining the length of the records in the file, and then close the file again. (The process of creating the file also opened it.)

Our example program uses a Keyed file made up of the Index file ADDRESS.KEY and the Data file ADDRESS.DAT. The program to create this Keyed file would be:

```
10 file%=1: recleng%=122 : lock%=2
20 CREATE #file%, "ADDRESS.DAT", "ADDRESS.KEY", lock%, recleng%
30 CLOSE #file%
40 END
```

Here `file%` is the file number (ie. the number that will be used to refer to the file for the remainder of this program) and `recleng%` is the length (in bytes) of the records we will be using in the file. The `lock%` may be set to 0, 1 or 2, corresponding to the different types of lock that can be applied to a file (see Part II). The lock we set here, 2, specifies exclusive access to the file and is the one we recommend using on single-user systems.

Type in this program, use it to create the new files (on the disc in the current default drive) and then save the program – for example as NEWADDR.BAS. You may need this program again while you are developing the main program: if there are any problems while you are testing the program, you may have to delete the Index file and Data file and then use your NEWADDR program to create a new Keyed file.

Working out the main program

The program will simply:

1. Open the address list file, define the record layout used in this file and allocate some memory for BASIC to manipulate the Keyed file in
2. Ask the user as many times as necessary to type A, R, D or Q to Add a record to the data base, Read the details of the record associated with a chosen key, Delete a record that's no longer wanted or Quit the program
3. Close the address list. •

This requires a very straightforward main program, using subroutines to:

- Add a new record to the file (at 3000)
- Read an existing record in the file (at 4000)
- Delete a record that is no longer required (at 5000)

The main program we need is:

```
100 '  
110 ' Preparatory stage  
120 ' =====  
200 PRINT "This is where we set aside some memory used for  
    manipulating Keyed files, open the Address file  
    and define the record layout"  
300 '  
310 ' Main program loop  
320 ' =====  
400 INPUT "Add, Read, Delete or Quit: Type initial letter ";  
        keyread$ : keyread$=UPPER$(keyread$)  
500 IF keyread$="A" THEN GOSUB 3000 ELSE IF keyread$="R"  
        THEN GOSUB 4000 ELSE IF keyread$="D" THEN GOSUB 5000  
600 IF keyread$ <> "Q" GOTO 300: REM Loop until "Q" is typed  
700 '  
710 ' Close Address file  
720 ' =====  
800 PRINT "This is where we close the Keyed file, ensuring  
    that the Index file and Data file on disc are correctly  
    up to date"  
900 END
```

We will now fill out this program with the details of the preparatory and closing stages and of the subroutines.

Preparatory stage

The first stage of any program that uses Keyed files must do three things:

1. Allocate some space in memory for manipulating Keyed files

The command to use is the BUFFERS command and the space you allocate sets the limit on the amount of the index that can be held in memory at any one time.

Space is available in 'buffers' of 128 bytes and the number of units you specify depends on whether you are more worried about the speed at which the program works or the amount of space available to your program as a whole. The more buffers you specify the faster Keyed files can be searched, but the space available to your program is reduced by 128 bytes per buffer. Six is usually a good number to start with, giving you the statement:

```
210 BUFFERS 6
```

2. Open the Keyed file you want to work with

The command we want here is:

```
OPEN "K", #file%, "ADDRESS.DAT", "ADDRESS.KEY",  
                                lock%, recleng%
```

This opens the existing Keyed file pair ADDRESS.DAT (the Data file) and ADDRESS.KEY (the Index file).

The command contains some special elements:

"K" which means that this is the Keyed version of the OPEN command

file% which is the number that will be used to refer to this Keyed file wherever it is used in this program

recleng% which holds the length of a name and address record

lock% which may be set to 0, 1 or 2, corresponding to the different types of lock that can be applied to a file: the lock we recommend you to use on single-user systems is 2 which gives you exclusive use of the file. (See Part II, Section 9.2 for details of the locks to use on multi-user systems.)

In our example, keyfile% will be 1 and recleng% will be 122 (20 for the name, five lots of 20 for the address and an extra 2 bytes that you always have to add to the total). These values stay the same throughout the program so you could write them directly into the OPEN command, but this would make the program harder to read and harder to update. It is better to have four statements:

```
220 file%=1: recleng%=122: lock%= 2  
230 OPEN "K", #file%, "ADDRESS.DAT", "ADDRESS.KEY",  
                                lock%, recleng%
```

3. Define how the records in this file are laid out

Before you can either read records or add new records to the data file, you have to use a command to define the layout of the records in the data file and assign string variables to each of these fields. This is called defining the record's 'Fields'. The Data file is a Random access file and so the command used to set up the fields in it is a FIELD command, exactly as you use a FIELD command to define the layout of records in a Random access file (see Chapter 6).

This FIELD command must always be after the OPEN command but before the loop which reads and writes the records.

For the address file we want the first 20 bytes for the name and then five lots of 20 bytes for the address, giving us the FIELD statement:

```
240 FIELD #file%, 20 AS namefld$, 20 AS addrfld1$, 20 AS  
      addrfld2$, 20 AS addrfld3$, 20 AS addrfld4$, 20 AS addrfld5$
```

Note how field variables all have individual names; they cannot be set up as members of an array.

With these commands, the first part of the program has now become:

```
100 '  
110 ' Preparatory stage  
120 ' =====  
130 '  
200 PRINT "This is where we set aside some memory used for  
      manipulating Keyed files, open the Address file and  
      define the record layout"  
210 BUFFERS 6  
220 file%=1: recleng%=122: lock%=2  
230 OPEN "K", file%, "ADDRESS.DAT", "ADDRESS.KEY",  
      lock%, recleng%  
240 FIELD #file%, 20 AS namefld$, 20 AS addrfld1$, 20 AS  
      addrfld2$, 20 AS addrfld3$, 20 AS addrfld4$, 20 AS addrfld5$  
300 '  
310 ' Main program loop  
320 ' =====  
330 '
```

Adding a record

The subroutine that adds a record to the data base (the subroutine at 3000) has to:

- Read in the new name and address from the keyboard
- Display these so that you can check them
- Fill the fields of the record with the data, and
- Write the record onto the disc

The actions of reading in a new name and address and displaying this information are standard data handling operations, not involving any of the special Jetsam commands at all. Moreover, they might well be required at a number of different points in the finished program, so we shall write these as separate subroutines – at 1000 and 2000, respectively.

Chapter 7: Keyed access files for data bases

In outline, this gives us the following subroutine:

```
3000 '
3100 ' Add a record to the data base
3200 '
3300 GOSUB 1000: REM - read in name and address
3400 GOSUB 2000: REM - display these to check they're OK
3500 INPUT "Is that correct? Type Y or N "; nameok$ :
                                nameok$=UPPER$(nameok$)
3600 IF nameok$="N" GOTO 3300:REM loop and ask again if not OK
3700 PRINT "This is where we fill the fields with the data"
3800 PRINT "This is where we add the record to the data base"
3900 RETURN
```

Before we go any further, it is a good idea to write the input and display subroutines at 1000 and 2000. If you have got these working, you can use them to test the Keyed file handling.

The input subroutine just needs to use INPUT statements to prompt you to type the name and then the lines of the address, and to read this information into the program. So the subroutine would be:

```
1000 '
1100 ' Subroutine to read name and address
1200 '
1300 INPUT "Name "; pername$
1400 FOR I%=1 TO 5
1500     INPUT "Address line "; peraddr$(I%)
1600 NEXT I%
1900 RETURN
```

Similarly, the display subroutine just needs to use PRINT statements to display the name and address information on the screen:

```
2000 '
2100 ' Subroutine to display name and address
2200 '
2300 PRINT pername$
2400 FOR I%=1 TO 5
2500     PRINT peraddr$(I%)
2600 NEXT I%
2900 RETURN
```

Note: These subroutines use an array – peraddr\$ – to store the five lines of the address. So we must remember to put a DIM statement at the beginning of the program as follows:

```
140 DIM peraddr$(5)
```

Filling the record fields with the data

The Data file is a Random access file and so we use the same techniques to fill the record fields with data.

Field string variables, as we explained when describing how to use Random access files, aren't given a value through a simple assignment statement like `addrfld2$ = "address"`. Instead the assignment takes one of the following three forms:

`LSET field-variable=string-expression`

`RSET field-variable=string-expression`

`MID$(field-variable, start, length) = string-expression`

LSET – which sets the field-variable to the string-expression, left aligned and padded out with spaces – is the one we shall use here. So we shall need six LSET commands as follows:

```
3710 LSET namefld$=pername$
```

```
3720 LSET addrfld1$=peraddr$(1)
```

```
3730 LSET addrfld2$=peraddr$(2)
```

```
3740 LSET addrfld3$=peraddr$(3)
```

```
3750 LSET addrfld4$=peraddr$(4)
```

```
3760 LSET addrfld5$=peraddr$(5)
```

Writing the record to disc

Once the record has been filled, we can add it to the data base. For this, we use a special Jetsam instruction – ADDREC which both puts the record into the Data file and puts the chosen key into the Index file.

The information we need to give ADDREC is the reference number of the Keyed file and the key we want to use for the entry:

```
ADDREC(#file%, 2, rank%, key$)
```

`file%` is the file number we are using in this program to identify the data file, 2 is (once again) the type of lock that we recommend you to use (exclusive access) and `key$` is the name the record is to be tagged with. As the index is kept in alphabetical order, `key$` also defines the place the key fits in the index. The other item in this instruction – `rank%` – is the number of the index you want to add the key to. You are allowed to put the keys associated with the records in the data base into up to eight indexes (known as ranks), identified by the numbers 0...7.

Where, as in this example case, you are only interested in keeping one index of these keys, it is simplest just to use rank number 0. Indeed, for most practical purposes, you can think of the ADDREC instruction as:

```
ADDREC(#file%, 2, 0, key$)
```

Chapter 7: Keyed access files for data bases

The key you use is up to you. The only restriction is that it is no more than 31 characters long. In a simple address file, the obvious thing to use to tag each record is the name, so we will use `pername$` as our key.

`ADDREC`, in common with the other special Jetsam instructions, is a function rather than a command – that is, it produces a value as well as performing the task. The Jetsam instructions are set up like this because in some cases BASIC won't be able to execute your instruction – for example, the command that deletes keys will fail if the key cannot be found. The value the instruction produces – known as the return code – gives you a way of checking that all is well. In general, a return code of zero indicates success, while a non-zero return code indicates failure. (In fact, the precise value tells you the reason for the failure – see Part II – but it is usually enough just to know whether the operation succeeded or failed.)

Thus the `ADDREC` statement we want is:

```
3810 rc%=ADDREC(#file%,2,0,pername$)
```

which we should follow with a test of whether the command was successful. To keep this example program simple, we will just display a message on the screen when the `ADDREC` fails – as follows:

```
3820 IF rc% <> 0 THEN PRINT "ADDREC FAILED,Return Code "; rc%
```

That completes the task of adding a record to the data base, making the completed subroutine:

```
3000 '
3100 ' Add a record to the data base
3200 '
3300 GOSUB 1000: REM - read in name and address
3400 GOSUB 2000: REM - display these to check they're OK
3500 INPUT "Is that correct? Type Y or N "; nameok$ :
                                nameok$=UPPER$(nameok$)
3600 IF nameok$="N" GOTO 3300:REM loop and ask again if not OK
3700 PRINT "This is where we fill the fields with the data"
3710 LSET namefld$=pername$
3720 LSET addrfld1$=peraddr$(1)
3730 LSET addrfld2$=peraddr$(2)
3740 LSET addrfld3$=peraddr$(3)
3750 LSET addrfld4$=peraddr$(4)
3760 LSET addrfld5$=peraddr$(5)
3800 PRINT "Add the record to the data base"
3810 rc%=ADDREC(#file%,2,0,pername$)
3820 IF rc% <> 0 THEN PRINT "ADDREC FAILED,Return Code "; rc%
3900 RETURN
```

Reading a record

Looking up a record in the data base is split into three stages:

- looking up the key in the index, which finds the record's record number but doesn't actually read the record
- getting the record from the Data file
- extracting and displaying the information in the record

In outline, this gives us the following for the subroutine at 4000:

```
4000 '  
4100 ' Read a record from the data base  
4200 '  
4300 PRINT "This is where we locate the record"  
4400 PRINT "This is where we get the record"  
4500 PRINT "This is where we extract the data"  
4600 GOSUB 2000: REM - display the data  
4900 RETURN
```

Looking up the Key in the Index file

This uses another special Jetsam instruction – `SEEKKEY(#file%, 2, rank%, key$)` which searches the Index file for the key you specify.

Again, `file%` is the file number we are using to identify the data file, 2 is the lock we suggest you set (exclusive access), `rank%` is the number of the index that you put the key in, and `key$` is the name you tagged the record with. In this simple program, we are just using one of the eight possible indexes – rank number 0 – so the instruction we want is `SEEKKEY(#file%,2,0,key$)`.

`SEEKKEY`, like `ADDREC`, is a function rather than a command, producing a return code so that you can see whether the operation was a success or a failure. Once again, a zero return code means success while a non-zero code indicates failure for some reason and the program needs to test whether the operation was successful. As with `ADDREC`, we will keep the program simple and just use the statements:

```
4320 rc%=SEEKKEY(file%,2,0,key$)  
4330 IF rc% <> 0 THEN PRINT "Failed to  
find ";key$, "Return Code ";rc%: RETURN
```

The only thing that is missing at this point are details of the key `SEEKKEY` is to search for. This has to be typed in, so the program needs a `INPUT` statement to prompt you to do this:

```
4310 INPUT "Type name to look up: "; key$
```


Note: BASIC won't be able to find the entry you require unless you type in the key using exactly the same combination of capital and lower case letters as was used when the key was set up. Extra trailing spaces after the key name will also stop the key from being found. However, your program could be designed to force all the keys into a particular style (eg. all capitals, no extra spaces) which will overcome this problem.

Getting the record from the data file

Using SEEKKEY looks up the Index file and finds the record number for the record in the Data file – rather like you might look up a topic in the index of a book and find out the number of the page it is described on. SEEKKEY doesn't actually read the record. Before you can read the information, you need to Get the relevant record from the file, just as you have to Get records from Random access files.

To get the record, we use the standard BASIC command GET.

This command generally needs to be told precisely which part of the file to get, but we can use the simple version of the command which gets the Current record. The SEEKKEY command we have just used changed the Current position to the key and record it found in the Index. So all we need here is:

```
4410 GET file%
```

Note: This simple form of the GET command is also available for getting information from Random files but then it gets the record following the current record, rather than the current record itself – a subtle but important difference.

Extracting and displaying the data

The GET command used above associates the information in the record with the field-variables defined in the FIELD statement at the beginning of the program. To use the information, you need to extract the data into standard program variables.

Extracting the data is like filling the fields with new information, but in reverse. But the process is simplified because ordinary assignment statements can be used:

```
4510 pername$=namefld$  
4520 peraddr$(1)=addrfld1$  
4530 peraddr$(2)=addrfld2$  
4540 peraddr$(3)=addrfld3$  
4550 peraddr$(4)=addrfld4$  
4560 peraddr$(5)=addrfld5$
```

To display the information, we can use the same subroutine that we used to help check that the information we were about to add to the Data file was correct – the subroutine at 2000, giving the line:

```
4600 GOSUB 2000: REM - Display the record
```

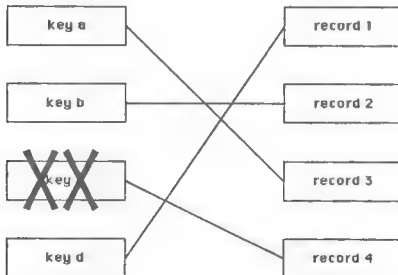
Thus the complete subroutine is:

```
4000 '
4100 ' Read a record from the data base
4200 '
4300 PRINT "This is where we locate the record"
4310 INPUT "Type name to look up: "; key$
4320 rc%=SEEKKEY(file%,2,0,key$)
4330 IF rc% <> 0 THEN PRINT "Failed to
                        find ";key$, "Return Code ";rc%: RETURN
4400 PRINT "This is where we get the record"
4410 GET #file%
4500 PRINT "This is where we extract the data"
4510 pername$=namefld$
4520 peraddr$(1)=addrfld1$
4530 peraddr$(2)=addrfld2$
4540 peraddr$(3)=addrfld3$
4550 peraddr$(4)=addrfld4$
4560 peraddr$(5)=addrfld5$
4600 GOSUB 2000: REM - Display the record
4900 RETURN
```

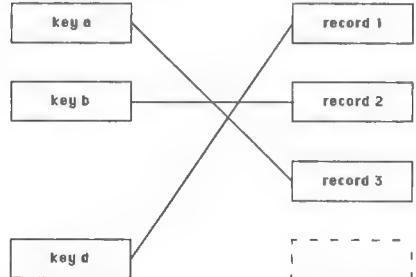
Deleting an entry

BASIC works its way around a Keyed file by finding the key in the Index file and then changing its Current position in the Keyed file to the combination of key and record it finds. If there is no key associated with a record, BASIC will never go to the record: it is as if the record does not exist. So the way to delete an entry from a Keyed file is simply to delete all the keys associated with the record.

Delete the key ...

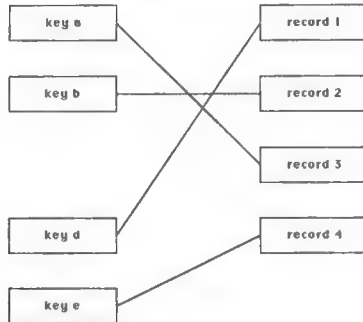


and the record is deleted too:



Chapter 7: Keyed access files for data bases

The data in the record initially remains in the file, but the next time a record is added to the data base, BASIC might well use that 'empty' slot to put the data into.



The subroutine that deletes an entry must:

- identify the record to be deleted, and then
- delete the key(s)

The simplest way to do this is to use our 'Read' subroutine to retrieve the record and then use a special Jetsam command – DELKEY – to delete the key.

The DELKEY command has two different forms: the simpler deletes the Current key (the Current position then moves to the next key and record in the index); the other deletes a specified key (leaving the Current position unchanged – unless, of course, you specified the Current key!). The effect of the 'Read' subroutine is to make the retrieved key the Current key, so we can use the simpler form of DELKEY:

```
rc%=DELKEY(#file%,2)
```

As usual, file% is the number we have associated with the Keyed file, 2 is the lock that we recommend and rc% is the return code produced by the Jetsam function to show whether the instruction was successful. However, in this case, a non-zero return code can mean success – up to 103. To avoid confusion in other parts of the program, the statement that checks whether the command was successful will also set all 'success' return codes to zero.

The subroutine we will use to delete the record is therefore:

```
5000 ' Delete a record from the data base
5200 '
5300 GOSUB 4000: REM - look up the record to delete
5400 IF rc%<>0 THEN RETURN: REM - exit if look-up failed
5500 rc%=DELKEY(#file%,2): REM - delete the current record
5600 IF rc%>103 THEN PRINT "Record not deleted" ELSE rc%=0
5900 RETURN
```

Closing the Keyed file

After you have finished with a Keyed file, you must close it. This does two things:

- ensures all the changes that you have made to the files are written to disc
- releases the record buffer and other resources for use with other files

If the files aren't properly closed, there is a risk that the Index file and the Data file stored on disc are inconsistent, with the result that the data base could give you incorrect results in future. Rather than let this happen, BASIC won't let you use a Keyed file unless you closed it properly the last time.

The command that does this is CLOSE and the statement we will use is:

```
810 CLOSE #file%
```

• That completes this simple address program. The complete program is shown below, together with the screen display from a short session of using the program.

```
100 '
110 ' Preparatory stage
120 ' =====
130 '
140 DIM peraddr$(5)
200 PRINT "This is where we set aside some memory used for
    manipulating Keyed files, open the Address file and
    define the record layout"
210 BUFFERS 6
220 file%=1: recleng%=122: lock%=2
230 OPEN "K", #file%, "ADDRESS.DAT", "ADDRESS.KEY",
                                lock%, recleng%
240 FIELD #file%, 20 AS namefld$, 20 AS addrfld1$, 20 AS addrfld2$,
    20 AS addrfld3$, 20 AS addrfld4$, 20 AS addrfld5$
300 '
310 ' Main program loop
320 ' =====
330 '
400 INPUT "Add, Read, Delete or Quit: Type initial letter
    (caps) "; keyread$: keyread$=UPPER$(keyread$)
500 IF keyread$="A" THEN GOSUB 3000 ELSE IF keyread$="R"
    THEN GOSUB 4000 ELSE IF keyread$="D" THEN GOSUB 5000
600 IF keyread$ <> "Q" GOTO 300: REM Loop until "Q" is typed
700 '
```

Chapter 7: Keyed access files for data bases

```
710 ' Close Address file
720 ' =====
730 '
800 PRINT "This is where we close the Keyed file, ensuring
      that the Index file and Data file on disc are up to date"
810 CLOSE #file%
900 END
1000 '
1100 ' Subroutine to read name and address
1200 '
1300 INPUT "Name "; pername$
1400 FOR I%=1 TO 5
1500     INPUT "Address line "; peraddr$(I%)
1600 NEXT I%
1900 RETURN
2000 '
2100 ' Subroutine to display name and address
2200 '
2300 PRINT pername$
2400 FOR I%=1 TO 5
2500     PRINT peraddr$(I%)
2600 NEXT I%
2900 RETURN
3000 '
3100 ' Add a record to the data base
3200 '
3300 GOSUB 1000: REM - read in name and address
3400 GOSUB 2000: REM - display these to check they're OK
3500 INPUT "Is that correct? Type Y or N "; nameok$ :
      nameok$=UPPER$(nameok$)
3600 IF nameok$="N" GOTO 3300: REM ask again if not OK
3700 PRINT "This is where we fill the fields with the data"
3710 LSET namefld$=pername$
3720 LSET addrfld1$=peraddr$(1)
3730 LSET addrfld2$=peraddr$(2)
3740 LSET addrfld3$=peraddr$(3)
3750 LSET addrfld4$=peraddr$(4)
3760 LSET addrfld5$=peraddr$(5)
3800 PRINT "This is where we add the record to the data base"
3810 rc%=ADDREC(#file%,2,0,pername$)
3820 IF rc% <> 0 THEN PRINT "ADDREC FAILED, Return Code "; rc%
3900 RETURN
4000 '
```

```
4100 ' Read a record from the data base
4200 '
4300 PRINT "This is where we locate the record"
4310 INPUT "Type name to look up: "; key$
4320 rc%=SEEKKEY(#file%,2,0,key$)
4330 IF rc% <> 0 THEN PRINT "Failed to find ";key$,
                                "Return Code ";rc%: RETURN
4400 PRINT "This is where we get the record"
4410 GET #file%
4500 PRINT "This is where we extract the data"
4510 pername%=namefld$
4520 peraddr$(1)=addrfld1$
4530 peraddr$(2)=addrfld2$
4540 peraddr$(3)=addrfld3$
4550 peraddr$(4)=addrfld4$
4560 peraddr$(5)=addrfld5$
4600 GOSUB 2000: REM - Display the record
4900 RETURN
5000 '
5100 ' Delete a record from the data base
5300 GOSUB 4000: REM - look up the record to delete
5400 IF rc%>0 THEN RETURN: REM - exit if look-up failed
5500 rc%=DELKEY(#file%,2): REM - delete the current record
5600 IF rc%>103 THEN PRINT "Record not deleted" ELSE rc%=0
5900 RETURN
```

Note: This program contains a number of PRINT statements that simply display a message about the part of the program that is about to be used. Such statements are useful when you are developing the program as a way of checking that control is being transferred around the program in the way you expect – especially when some parts of the program exist only in outline. When the program is fully working, convert these statements into comments – by inserting ' at the beginning of each statement.

Enhancements

Many desirable features have been left out of this program. It doesn't allow you to change an entry in the data base if, for example, someone moves house. It also doesn't allow you to change the key that you are using for an entry, or to index records under a number of keys, and if you had tagged more than one entry with the same key (two Mr Smiths, say), it would only ever find the one that has been in the data base longest.

To illustrate how to give your program these features, we will now prepare appropriate subroutines for each of these tasks.

Adding extra keys

Each key in the Index file is associated with just one record but each record can be tagged with any number of keys, either in the same index or in different indexes.

The ADDREC instruction you use when you add the record to the data base lets you tag the record with one key. Further keys are added one at a time with ADDKEY instructions. These extra keys can be added to any one of the eight indexes or 'Ranks' that are available in each Index file. They don't have to be added to the same Rank as the original key.

Before you can use ADDKEY to add extra keys, you need to know the record's record-number. If your program adds these extra keys immediately after the ADDREC that wrote the record, the record you want will be the Current record and you can use another Jetsam instruction – the FETCHREC instruction – to discover its record-number. When your program is going back to information that was stored earlier, you need to retrieve the record first (making it the Current record) and then use FETCHREC to tell you its record-number. (If in doubt, retrieve the record, because then you can check that you are picking out the right information.)

If we were to add such a subroutine at 6000, it would be:

```
6000 '  
6010 ' Subroutine to add extra keys to one record  
6020 '  
6100 GOSUB 4000: REM - retrieve record using existing  
      subroutine, which also displays the retrieved information"  
6200 rec.no = FETCHREC(#file%): REM - remember record number  
6300 INPUT "Type the new key. If no more keys to add,  
      just press RETURN "; newkey$  
6310 IF newkey$="" THEN RETURN: REM - exit subroutine  
      when no more keys to add  
6400 INPUT "Type the number of the index you want to add  
      the key to (0..7) "; rank%  
6500 rc% = ADDKEY(#file%,2,rank%,newkey$,rec.no): REM  
      - add the new key in the specified index  
6510 IF rc% >= 130 THEN PRINT "ADDKEY failed. Operation  
      abandoned": RETURN: REM - exit subroutine  
      as ADDKEY unable to add to this record  
6520 IF rc% <> 0 THEN PRINT "Key not acceptable":REM see Pt II  
6600 GOTO 6300: REM - loop to add further keys  
6900 RETURN
```

The operation of this subroutine should be clear from the comment statements. As usual, file% is the number we are using to refer to the Keyed file and 2 is the lock we recommend.

Changing the key

To change the key that a record has been tagged with, you have to both add a new key (using ADDKEY) and get rid of the old key (using DELKEY).

The order in which you do these operations is important. You must add the new key before you delete the old key. If you delete the old key first and it is the only key, you will delete the record as well. We earlier used precisely this effect to delete records from the data base!

The subroutine we need uses very much the same statements as the subroutines to Add an extra key and to Delete a record. In fact, you could just make your program call first the Add key subroutine and then the Delete one. However, this wouldn't be a very good solution because each subroutine would start by using the Read a record subroutine to search the Index for the same key. A better program would search for the key once and remember the details it finds.

To keep the program simple, we will just have a subroutine that changes a key in the first index (rank 0):

```

7000 '
7010 ' Subroutine to Change a key
7020 '
7100 GOSUB 4000: REM - look up record to change
7200 rec.no = FETCHREC(#file%): REM - remember record number
7300 INPUT "Type the new key "; newkey$
7400 rc% = ADDKEY(#file%,2,0,newkey$,rec.no): REM - add the
                                new key in the specified index
7500 IF rc% = 0 THEN rc% = DELKEY(#file%,2,0,key$,rec.no): IF
                                rc%<=130 THEN rc% = 0 : REM - Delete old key and tidy rc%
7600 IF rc% <> 0 THEN PRINT "Changing key value failed,
                                Return code = "; rc%
7900 RETURN

```

Notice the form of the DELKEY instruction we have used here – DELKEY(#file%,2,0,key\$,rec.no). This has more parameters than the DELKEY instruction we used earlier – the index we are working on (rank 0), the old key and the record-number the key points to. We have to specify both the key and the record-number as there could be two different records that are tagged with the same key – for example, two different Mr Browns.

We have had to use the longer form of the DELKEY command because the key we want to delete is no longer the Current key because the ADDKEY command moved the Current key to the new key. If we had used the simpler form of DELKEY, we would have deleted the key we had just added!

Changing an entry in the data base

The simplest sort of change to make is to change the details of the record without affecting the key the record is tagged with – the kind of change involved when someone moves house. The subroutine that makes such a change has to:

- retrieve the current version of the record
- replace the information in the record fields with the new information, and
- save the new information to disc

In outline, the subroutine we need is:

```
8000 '
8010 ' Change a record in the data base
8020 '
8100 GOSUB 4000: REM - retrieve the relevant record and read it
8200 '
8210 ' Read in the new information and display it
8220 '
8230 GOSUB 1000: REM - read in the new information
8240 GOSUB 2000: REM - display it to check it's OK
8250 INPUT "Is that correct? Type Y or N "; nameok$ :
                                nameok$=UPPER$(nameok$)
8260 IF nameok$ = "N" GOTO 8230:REM loop and ask again if OK
8300 '
8310 ' Set up the new record fields
8400 '
8410 ' Save the new information to disc
8420 '
8900 RETURN
```

The process of setting up the new record fields uses the same commands as we used when adding a completely new record to the data base:

```
8330 LSET namefld$=pername$
8340 LSET addrfld1$=peraddr$(1)
8350 LSET addrfld2$=peraddr$(2)
8360 LSET addrfld3$=peraddr$(3)
8370 LSET addrfld4$=peraddr$(4)
8380 LSET addrfld5$=peraddr$(5)
```

As we are just changing information stored in the record, we can use a PUT command to write the new information to disc. Like GET, this command usually needs to be told whereabouts on the disc the information has to be written, but as the record in question is the Current record we can use the simple form:

```
8430 PUT #file%
```

Note: This simple form of the PUT command is also available for writing information in Random files but then it writes the record following the current record, rather than the current record itself – a subtle but important difference.

The complete subroutine is therefore

```
8000 '
8010 ' Change a record in the data base
8020 '
8100 GOSUB 4000: REM - retrieve the relevant record and read it
8200 '
8210 ' Read in the new information and display it
8220 '
8230 GOSUB 1000: REM - read in the new information
8240 GOSUB 2000: REM - display it to check it's OK
8250 INPUT "Is that correct? Type Y or N "; nameok$ :
                                nameok$=UPPER$(nameok$)
8260 IF nameok$ = "N" GOTO 8230:REM loop and ask again if OK
8300 '
8310 ' Set up the new record fields
8320 '
8330 LSET namefld$=pername$
8340 LSET addrfld1$=peraddr$(1)
8350 LSET addrfld2$=peraddr$(2)
8360 LSET addrfld3$=peraddr$(3)
8370 LSET addrfld4$=peraddr$(4)
8380 LSET addrfld5$=peraddr$(5)
8400 '
8410 ' Save the new information to disc
8420 '
8430 PUT #file%
8900 RETURN
```

Changing and re-indexing an entry

The more difficult case of changing a record is when this changes the key as well as the information. For example, you might want to update your name and address list when someone gets married, changing both name (the key) and address. In such cases, there are two approaches:

- 1 Change the data in the record, then re-index
- 2 Delete both the record and the key, and add the new details from scratch

The second of these two approaches is simpler and easier to write. We could program it as follows:

Chapter 7: Keyed access files for data bases

```
9000 ' Change the record and key
9010 '
9100 GOSUB 5000: REM - look up and delete old record
9110 IF rc%<>0 THEN RETURN: REM - exit if look up fails
9200 GOSUB 3000: REM - add new information
9900 RETURN
```

But this approach won't always work. One case when it fails is when there is more than one key indexing a particular record.

A subroutine that takes Approach 1 would cope with multiple keys and could be combined with the subroutine we prepared above that changed just the record in the disc. You just need to remember the old key and see whether it changes when you acquire the new information. If it does, then you need to change the key as well – by using the subroutine we have prepared at 7000 to change the key (ideally modified so that the program doesn't ask for the new key twice!)

In outline, the combined subroutine would be:

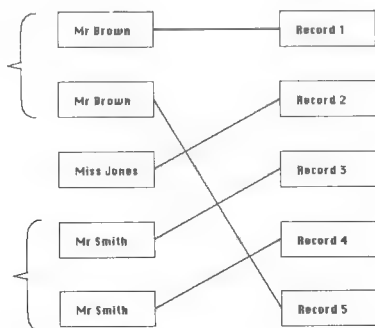
```
8010 ' Change a record in the data base
8020 '
8100 GOSUB 4000: REM retrieve the relevant record and read it
8110 old.name$=key$:REM - remember the old name (current key)
8200 '
8210 ' Read in the new information and display it
8220 '
8230 GOSUB 1000: REM - read in the new information
8240 GOSUB 2000: REM - display it to check it's OK
8250 INPUT "Is that correct? Type Y or N "; nameok$ :
      nameok$=UPPER$(nameok$)
8260 IF nameok$ = "N" GOTO 8230:REM loop and ask again if OK
8300 '
8310 ' Set up the new record fields
8400 '
8410 ' Save the new information to disc
8500 '
8510 ' Test whether name has changed; if so change key
8520 '
8530 IF old.name$<>pername$ THEN GOSUB 7000:REM change key
8900 RETURN
```

The point to note about this subroutine is the way it updates the information in the record before it changes the key. Whenever you want to change both the information in the record and any keys, you must be careful to finish changing the information before you start changing the keys. You mustn't have any ADDKEY or DELKEY instructions between the GET statement that reads the old information in the file and the PUT statement that overwrites the record with the new information.

Selecting among identical keys

The simple Read a record subroutine at 4000 asks for a key and gives you the details of the entry it finds. It doesn't give you the chance to pick out other entries that have been tagged with the same key.

When you tag a new record with the same key as an existing record, the new key is inserted into the Index immediately after the existing one. This means you can gradually build up an Index containing 'sets' of identical keys, with the oldest entry with this key at the top of the set and the newest at the bottom. For example, in an address list that included a number of Mr Smiths and a number of Mr Browns, you would have a set all with the key 'Mr Smith' and a set with the key 'Mr Brown'.



The problem with the simple Read routine is that it can only ever pick out the entry at the top of the set with this key. A better subroutine would allow you to reject this record and look at the next one in the index, so that you could gradually work through all the Mr Smiths until you find the one you want.

The Jetsam command needed here is SEEKNEXT. Once SEEKKEY has been used to find the first entry in the set, SEEKNEXT can be used to move to the next entry in the Index – which will either have the same key or the next key in the alphabetical list. (BASIC also has instructions to move to the previous entry in the Index – SEEKPREV – and to move to the next entry with a different key – SEEKSET.)

The modified subroutine should ask whether the record that is displayed is the one they wanted and, if it isn't, whether the user wants to continue searching with the same key or with a different key. If the search is to continue with the different key, the program just loops back to beginning of the subroutine – to start all over again. But if the search is to continue with the same key, the program uses SEEKNEXT to move the Current position to the next entry in the Index and then loops back to the part of the subroutine that Gets and displays the Current record.

Chapter 7: Keyed access files for data bases

The subroutine then becomes:

```
4000 '
4100 ' Read a record from the data base
4200 '
4300 PRINT "This is where we locate the record"
4310 INPUT "Type name to look up: "; key$
4320 rc%=SEEKKEY(#file%,2,0,key$)
4330 IF rc% <> 0 THEN PRINT "Failed to find ";key$,
                                "Return Code ";rc%: RETURN
4400 ' This is where we get the record
4410 GET #file%
4500 ' This is where we extract the data
4510 pername$=namefld$
4520 peraddr$(1)=addrfld1$
4530 peraddr$(2)=addrfld2$
4540 peraddr$(3)=addrfld3$
4550 peraddr$(4)=addrfld4$
4560 peraddr$(5)=addrfld5$
4600 GOSUB 2000: REM - Display the record
4700 INPUT "Is this the right name and address? Y or N ";
                                nameok$ : nameok$=UPPER$(nameok$)
4710 IF nameok$="Y" THEN RETURN: REM - exit subroutine if
                                right record found
4800 ' Determine how to continue search
4810 INPUT "Continue search with this name as the key or Try
                                new key. Type C or T (or any other letter for
                                neither)"; cont$ : cont$=UPPER$(cont$)
4820 IF cont$="T" GOTO 4300: REM - start again with a new key
4830 IF cont$="C" THEN rc%=SEEKNEXT(file%,2): IF rc% = 0 GOTO 4400
                                ELSE PRINT "SEEKNEXT failed. No more entries
                                for this name": GOTO 4800
4900 RETURN
```

As with the other Jetsam commands, SEEKNEXT produces a Return code so that you can see whether the command was successful. In this case, Return codes below 103 are all successes but the program only tests for a Return code of 0 because this tells you that the entry that has been found has the same key.

Example

The Task: The task is to provide a 'phone book' rather like the one we used in Chapter 6 to illustrate the use of sequential files, but with the following extra features:

- up to 1000 entries
- one or more phone numbers for each client
- production of phone lists in alphabetical order of name (name, phone number)
- production of phone lists by client type (name, phone number)

There are three client types – 'A', 'B' and 'C'.

If you want to test your understanding of the facilities described so far, attempt to design and write the program now, without reading any further.

The design As the number of entries is large, the whole file cannot be read into an array in memory. The need to be able to access the phone book in two different ways (by name and by client type) indicates that a keyed file will be better than a random access file.

Part of the design is deciding the way that the information will be stored and keyed. The information to be stored is :

- Name
- Phone number
- Client type

The ways in which information will be requested are:

- by name
- by client type

You will need a different key for each way that information will be requested, so two ranks will be needed – one for names, one for client types. Is there any need to store the information in the data file too? This depends on whether this information will ever be needed when another key has been used to access the record and whether there is room in a key for the full data. In this example, the information will be stored as keys and in the data file.

To make the program somewhat easier to use, keys will be restricted to the first six characters of the name, so the user need only know the first part of the client name to find a number.

The program will consist of the following major stages:

- set up variables and screen
- open or create phone book file
- ask for and carry out instructions (search by name, add an entry, finish)
- close file and tidy up

The following description presents one way of translating these stages into a program. Many of these steps have already been described previously. It is left as an exercise for the reader to translate this text into a program.

- Prepare variables and screen.
- If data and index files are present, open the keyed file.
- If data and index files are not present ask whether to change disc or create a new file and do so.
- Display options, – Search, Add, List by Name, List by Client type, Finish.
- Get and obey user options, as follows:

Search:	<ul style="list-style-type: none">– obtain name, shorten it to key length, case-convert and use as key– SEEKKEY in names rank with this key– If not found, say so and ask for another name– If found, read record and show it– Ask if want to look for another– If yes, SEEKNEXT– If found in same set, read record, show it and ask another?– If not found, say so and ask for another name– If no, return
Add:	<ul style="list-style-type: none">– obtain name, phone number, client type– shorten name to key length, case-convert and use as key– ADDREC with name key to names rank– ADDKEY client type to client types rank– return
List by Name:	<ul style="list-style-type: none">– SEEKRANK names rank– read record and print it– SEEKNEXT– if same rank, print it and repeat– if different rank, return
List by Client type:	<ul style="list-style-type: none">– ask client type– SEEKKEY with client type name specified– read record and print it– SEEKNEXT if same key, print it and repeat– if different key set rank, return
Finish:	<ul style="list-style-type: none">– close file

Machine level operations

Computer magazines and other such publications often print details of 'Machine code programs' and 'Patches' that you can make in order to give your computer system special facilities. For example, you may see a short machine code program that someone has written that will let you 'read' the date and time recorded by the clock built into your computer, or you might see details of a Patch that will let you modify the characters displayed on the screen.

The design of machine code programs and patches requires a lot of detailed technical knowledge about your computer and about the software you are using on it. Such technical detail is outside the scope of this manual: for that sort of information you will need to turn to a book on Machine-code or Assembler programming on the microprocessor in your computer. The most we give here is details of how the microprocessor's registers will be set on entry to the machine code routine and how BASIC expects these to be set when the program returns from this routine (given in Part II, Appendix III).

However, making use of published programs and patches is really very straightforward and this is what we describe in this chapter.

General information

All the software to run your computer is held in your computer's memory, as an array of individual codes. Each code takes up one byte of computer memory and has its own 'address' so that it can be readily found.

When you switch on your computer and load the operating system, all the routines that make up the operating system are read into one area of the computer's memory – usually at the 'top' of the memory (high-numbered addresses). Similarly, when you load BASIC, the software that makes up the BASIC programming language is read into a separate area of memory. Any BASIC program you want to load and the variables this program generates are held in a third area, which BASIC automatically reserves as its working area.

'Patches' change the codes at particular locations in the memory. This has the effect of changing a piece of information or how a particular routine works, for example

causing the program to 'jump' to a particular routine. The published machine code programs usually contain extra routines that you might want to make use of.

Applying a patch or making an machine code program available both involve writing new codes at particular locations in memory. In the case of a patch, these locations are predefined by the information that you want to change. In the case of a machine code program, you will probably write the new codes one after another into an empty area of the memory. Normally the easiest place to put it is at the top of the area BASIC reserves as its working area (your programs rarely need to use all of this area). You then have to arrange that the piece of memory you have used is no longer treated as part of BASIC's working area. (This is in fact essential if you will be calling the routine from BASIC.)

The result of any patch will appear automatically whenever the affected code is accessed: you don't have to do anything special to cause the patch to be used. But you will have to direct your computer to execute the codes of the additional routine and that means remembering the address of the first code because this is the entry point into the program.

So to use published patches and machine-code programs, you need to be able to:

- write new codes at specific locations anywhere in its memory;
- divide off a section of memory from BASIC's working area; and
- direct the program to the start of an extra routine

Commands that do each of these actions are available within BASIC.

Applying a Patch

The BASIC command used to patch bytes in memory is the POKE command, `POKE address, new-code`. This writes the new code at the address given in the command, replacing whatever code was there before. So you just need a separate POKE command for each byte you want to change.

The address needed by the POKE is the location of the byte: this can be any integer between 0 and 65535. The new code is also expressed as an integer, and as the code occupies one byte, this can take any value between 0 and 255.

The addresses and the new codes form the details of the patch; they will probably be written either as decimal numbers or as hexadecimal numbers. Either form is equally acceptable in the POKE command. You just have to copy these details into POKE commands – indeed, they may even be already presented as POKE commands. These POKE commands can then either form a short BASIC 'Set-up' program or be included in another BASIC program.

The following short program illustrates such a short 'Set-up' program. This is in fact the preparation program that users of Amstrad PCW computers need to run before using the routines of GSX, the Graphics Extension to CP/M. It sets the 'jump' to GSX.

```

100 GSX%=&H30
110 POKE GSX%+0, &H50 'LD D,B
120 POKE GSX%+1, &H59 'LD E,C
130 POKE GSX%+2, &HE 'LD C, 115
140 POKE GSX%+3, 115
150 POKE GSX%+4, &HC3 'JP &H0005
160 POKE GSX%+5, &H5
170 POKE GSX%+6, &H0

```

Loading a machine code program

Loading a machine code program is very like applying a patch, but with a few extra steps that reserve the area of memory for it. The stages involved are therefore:

- work out how much memory is required and divide this area from the memory reserved by BASIC
- load in the codes of the routine

Allocating the memory

The amount of memory required for the routine is defined by the number of codes in the routine. For example the following routine (which will fetch the time when you are running BASIC under a DOS operating system) needs 29 bytes to store it:

```

&HB4 &H2C &HCD &H21 &H8B &HEC &H8B &H5E
&H08 &H8A &HC5 &H98 &H89 &H07 &H8B &H5E
&H06 &H8A &HC1 &H89 &H07 &H8B &H5E &H04
&H8A &HC6 &H89 &H07 &HCB

```

This memory then needs to be taken off the top of BASIC's working area. To do this, you use two more of BASIC's commands and functions – the HIMEM function to tell you the current address of the top byte in this area of memory, and the MEMORY command to move the top of the area to another address. In fact, you will typically put these together into the one instruction `MEMORY HIMEM-bytes`; so for example, to divide off a section of memory for the time routine given above, you would use:

```
MEMORY HIMEM-29
```

As the whole point of loading an external routine is to use it, the area used to store an external routine isn't re-absorbed into BASIC's working area at the end of the program. However, this means that any program that carves off a section of memory from the working area should only be run once per session – or, if it does need to be run again, the corresponding `MEMORY HIMEM+bytes` command should be used first. Otherwise, the working area will be gradually whittled away because the `HIMEM` isn't reset to its original value until you load BASIC again.

Note: When you have more than one external routine or function you want to use, you need to divide off an area from BASIC's working area for each routine. This can be done either in one program or in a number of different programs – for example, one program for each routine. Each of these programs would have its own `MEMORY` instruction. Of course, the order in which you execute these programs will affect where each routine is actually stored in memory and so it is a good idea to keep careful track of this.

Loading the routine

The process of loading the routine simply involves executing a series of `POKE` commands, which write the individual codes of the routine into memory – starting at the byte immediately above the new `HIMEM`.

Because the routine is to be inserted into consecutive bytes, these `POKE` commands can readily be handled by a `FOR...NEXT` loop. For example, the following could be used to read the time routine from a number of `DATA` statements:

```
100 FOR i = 1 TO 29
110 READ j
120 POKE HIMEM+i, j
130 NEXT
150 DATA &HB4, &H2C, &HCD, &H21, &H8B, &HEC, &H8B, &H5E
160 DATA &H08, &H8A, &HC5, &H98, &H89, &H07, &H8B, &H5E
170 DATA &H06, &H8A, &HC1, &H89, &H07, &H8B, &H5E, &H04
180 DATA &H8A, &HC6, &H89, &H07, &HCB
```

As well as listing the codes, the designer will normally give the total sum of the codes because this provides a simple check of whether you have typed all the codes into your program correctly. This total is known as the checksum.

The checksum for the time routine is 3343. You could put this into your routine as follows:

```
90 jsum=0
100 FOR i = 1 TO 29
110 READ j : jsum=jsum+j
120 POKE HIMEM+i,j
130 NEXT
140 IF jsum<>3343 THEN PRINT "Error in DATA statements" : STOP
150 DATA &HB4, &H2C, &HCD, &H21, &H8B, &HEC, &H8B, &H5E
160 DATA &H08, &H8A, &HC5, &H98, &H89, &H07, &H8B, &H5E
170 DATA &H06, &H8A, &HC1, &H89, &H07, &H8B, &H5E, &H04
180 DATA &H8A, &HC6, &H89, &H07, &HCB
```

Using a machine code program

The way you use a machine code program that you've loaded into memory depends on whether it forms a routine (a package of instructions) or a function (a packaged expression).

Using an external routine

To use an external routine, you CALL it, giving the address of the entry point to the routine and the names of any variables you need to pass between the routine and the rest of your program.

The entry point to the routine is often the address of its first byte, ie. HIMEM+1. To pass this address to the CALL command, you first record this value as a variable and then use this variable in the CALL instruction. For example, you might record the entry point to the time routine as `get.time=HIMEM+1`.

The variables you pass are defined by the routine itself. For example, the time routine requires three integer variables, into which it will put the hour, the minute and the second. These variables can then be used in the rest of your program: for example, you might decide simply to print them out as follows:

```
200 get.time=HIMEM+1
210 CALL get.time(hour%,min%,secs%)
220 PRINT "Time is now "; USING "##:##:##"; hour%, min%, secs%
```

Using an external function

To use an external function, you first have to define it but then you can use it just like any other function.

The process of defining it associates the function with one of the 10 names reserved for external functions, USR0...USR9. The command used to do this is DEF USR, and again you need to give the entry point to the routine (the address of the first byte).

Suppose the machine code program you had just poked into memory formed an external function. Then the following instructions could be used to define it as the external function USR0:

```
200 address=HIMEM+1
210 DEF USR0=address
```

To use this function, you might then have some instruction like:

```
220 result=USR0 (value)
```

where `result` is the result of the function USR0 on `value`.

Examples

The following are examples of BASIC programs which use the techniques described in this chapter. You won't necessarily be able to use these specific programs on your computer – the Date and Time programs, for example, can only be used on a computer running MS-DOS, PC-DOS or DOS Plus. Their role is more to provide useful models of the kind of program you might write in order to make use of other external programs.

Of course, if you are using the appropriate computer and operating system, you might wish to make use of these programs directly.

Example 1: Getting the date and time from DOS

The following programs show how it is possible to fetch the current date and time from a Mallard BASIC program running under MS-DOS, PC-DOS or DOS Plus.

The first part of each program sets up a machine code routine to call the operating system, asking for details of the date or the time as appropriate: the second part of the program shows the way in which the machine-code routine should be used. The hexadecimal numbers in the DATA statements are the machine code of the routines themselves.

The machine code routines are stored in some of the memory which was initially allocated to BASIC (reserved by the MEMORY HIMEM-bytes statements).

If you are running BASIC under MS-DOS, PC-DOS or DOS Plus, you can use these programs by just typing them in and running them. You can use upper or lower case letters and you don't have to type the REM statements if you cannot be bothered, but almost any other difference is likely to be significant. In particular, the DATA statements must be typed *exactly* as shown: otherwise the machine may lock up as you try to run the program. Indeed, it is wise to save each program before running it because otherwise you will have to type it in again if the machine does lock up.

The programs are designed to be run just once per session. If you need to run them again, remember to return the area of memory reserved for the machine code to BASIC first, otherwise the memory will be gradually whittled away.

The programs simply use the routines once and print the results. Clearly there are many more useful things that you might do, for example with extra CALLs to the machine code routine. If you wish, you can use the two routines together in the same program. Of course, CALLs must be in exactly the form given, with four DATE or three TIME parameters – all integers.

```

100 REM
110 REM Fetching the time
120 REM
130 REM Lines 210...290      Set up the machine code and
140 REM                      should only be executed once
150 REM Lines 310...320     Give an example of use
160 REM NB: You must give all the parameters in the
170 REM order given even if you will only use some of
180 REM them. You can change their names if you wish but
190 REM they must be integer variables
200 REM
210 MEMORY HIMEM-29
220 jsum=0
230 FOR i = 1 TO 29 :READ j : jsum=jsum+j :POKE HIMEM+i,j : NEXT
240 IF jsum<>3343 THEN PRINT "Error in DATA statements" : STOP
250 get.time = HIMEM+1
260 DATA &HB4, &H2C, &HCD, &H21, &H8B, &HEC, &H8B, &H5E
270 DATA &H08, &H8A, &HC5, &H98, &H89, &H07, &H8B, &H5E
280 DATA &H06, &H8A, &HC1, &H89, &H07, &H8B, &H5E, &H04
290 DATA &H8A, &HC6, &H89, &H07, &HCB
300 REM
310 CALL get.time(hour%, min%, secs%)
320 PRINT USING "& ##:##:##";"Time is now ";hour%,min%,secs%
330 END

```

```
100 REM
110 REM Fetching the date
120 REM
130 REM Lines 210...300      Set up the machine code and
140 REM                      should only be executed once
150 REM Lines 320...410     Give an example of use
160 REM NB: You must give all the parameters in the
170 REM order given even if you will only use some of
180 REM them. You can change their names if you wish but
190 REM they must be integer variables
200 REM
210 MEMORY HIMEM-33
220 jsum=0
230 FOR i=1 TO 33 :READ j : jsum=jsum+j :POKE HIMEM+i, j :NEXT
240 IF jsum <> 3554 THEN PRINT "Error in DATA statements":STOP
250 get.date=HIMEM+1
260 DATA &HB4, &H2A, &HCD, &H21, &H8B, &HEC, &H8B, &H5E
270 DATA &H0A, &H98, &H89, &H07, &H8B, &H5E, &H08, &H8A
280 DATA &HC2, &H98, &H89, &H07, &H8B, &H5E, &H06, &H8A
290 DATA &HC6, &H89, &H07, &H8B, &H5E, &H04, &H89, &HOF
300 DATA &HCB
310 REM
320 wday$(0)="Sunday"
330 wday$(1)="Monday"
340 wday$(2)="Tuesday"
350 wday$(3)="Wednesday"
360 wday$(4)="Thursday"
370 wday$(5)="Friday"
380 wday$(6)="Saturday"
390 REM
400 CALL get.date(wday%,day%,month%,year%)
410 PRINT USING "& & ##/##/####"; "Today is ";
                                wday$(wday%), day%, month%, year%
420 END
```

Example 2: Using GSX routines

GSX is the Graphics Extension to CP/M and GSX routines can be used to generate a wide range of graphics effects, though not all of these are available on all computers that run CP/M with the Graphics extension. Details of these routines and how they are called are outside the scope of this user guide: they are given in Digital Research's GSX manual.

GSX routines are available on a number of different computer systems. The instructions given here are the ones for use on the Amstrad PCW machines.

Essential preparation

To use GSX routines from BASIC, you first need to produce a copy of BASIC with GSX installed on it and to set up the ASSIGN.SYS file so that it reflects the device drivers that you will actually be using.

To do this, you should first create a disc with copies of the following files on it:

- BASIC.COM (the file holding Mallard BASIC on Amstrad PCW system discs)
- GENGRAF.COM
- GSX.SYS
- ASSIGN.SYS
- and the relevant device drivers (with names like DDSCREEN.PRL)

You should rename BASIC.COM to GSXBASIC.COM, and then attach GSX to this copy by using the command:

```
GENGRAF GSXBASIC.COM
```

GENGRAF.COM can then be removed from the disc.

Finally, modify the ASSIGN.SYS file so that it reflects the actual devices to be used. (Your computer's CP/M guide should explain how to do this.) It is wise to remove from the ASSIGN.SYS any devices which you do not intend to use. **Note:** All the device drivers used by ASSIGN.SYS must be present on your disc.

Running BASIC and GSX together

To use GSX from BASIC, you have to load the version of BASIC to which GSX has been attached and then set the jump to GSX.

The version of BASIC you require is the stored as GSXBASIC.COM, so to run it you just have to insert the correct disc and type the command: GSXBASIC

To set the jump, you need to run the the following program:

```
100 GSX%=&H30
110 POKE GSX%+0, &H50 'LD D,B
120 POKE GSX%+1, &H59 'LD E,C
130 POKE GSX%+2, &HE 'LD C, 115
140 POKE GSX%+3, 115
150 POKE GSX%+4, &HC3 'JP &H0005
160 POKE GSX%+5, &H5
170 POKE GSX%+6, &H0
```

(We suggest saving this program as GSXPREP, as you will need it each time you run GSXBASIC.)

The program

A BASIC program that uses GSX has to contain the following, at some point before the first call of GSX:

```
DIM contrl%(6),ptsin%(128),ptsout%(12),intin%(128),  
                                intout%(45):gsx%=&H30
```

(The arrays given here use the same names as those described in the GSX Programmer's Guide. Other names can be used but they must be integer arrays. Note that the parameter block (PB) is not explicitly declared.)

To invoke a GSX function, the program has to set suitable values into the contrl% array and put suitable parameters into the ptsin% and intin% arrays. GSX is then called by:

```
CALL gsx%(gsx%,gsx%,contrl%(1),intin%(1),ptsin%(1),  
                                intout%(1), ptsout%(1))
```

The program given here will draw a filled area on PCW9000 series machines, but on PCW8000 series machines this area won't be filled unless the enhanced versions of the screen drivers are being used. (Locomotive Systems can supply the enhanced drivers on a 3" disc for PCW owners (together with example programs). The enhanced drivers are also supplied with copies of Digital Research's GSX programs DR Draw and DR Graph.)

```
100 GOSUB 60000 ' Set up for GSX Calls  
110 PRINT "Number of points in shape (between 3  
                                and 32)"; : INPUT n  
120 IF n<3 OR n>32 THEN PRINT "Must be between 3 and 32  
                                (inclusive).":GOTO 110  
130 GOSUB 60100 ' Open workstation  
140 contrl%(1)=23 : intin%(1)=2+RND : GOSUB 61100 ' set  
                                fill interior style  
150 contrl%(1)=24 : intin%(1)=1+5*RND :  
                                GOSUB 61100 ' set fill style index  
160 FOR i=1 TO n : ptsin%(2*i-1)=32000*RND :  
                                ptsin%(2*i)=32000*RND : NEXT  
170 contrl%(1)=9 : contrl%(2)=n : contrl%(4)=0 : GOSUB 61000  
180 PRINT "Again (Y/N)"+CHR$(27)+"e";  
190 r$=UPPER$(INKEY$) : IF r$<>"N" AND r$<>"Y" THEN 190  
200 GOSUB 60200 ' Close workstation  
210 IF r$="Y" THEN 110  
220 PRINT "Finished"  
230 END
```

```

60000 '
60010 ' Define arrays and address of "jump" to GSX
60020 '
60030 gsx%=&H30
60040 DIM contrl%(6),ptsin%(128),ptsout%(12),intin%(128),
                                                intout%(45)
60050 RETURN
60100 '
60110 ' Open Workstation: Device = 1
60120 '
60130 contrl%(1)=1 : contrl%(2)=0 : contrl%(4)=10
60140 FOR i=1 TO 10 : intin%(i)=1 : NEXT
60150 GOSUB 61000
60160 contrl%(1)=14 : contrl%(2)=0 : contrl%(4)=4
60170 intin%(1)=1 : intin%(2)=1000 : intin%(3)=1000 :
                                                intin%(4)=1000
60180 GOTO 61000 ' Set Colour Index 1 = White
60200 '
60210 ' Close Workstation
60220 '
60230 contrl%(1)=2 : contrl%(2)=0 : contrl%(4)=0 : GOTO 61000
61000 '
61010 ' Actual call of GSX
61020 '
61030 CALL gsx%(gsx%,gsx%,contrl%(1),intin%(1),ptsin%(1),
                                                intout%(1), ptsout%(1))
61040 RETURN
61100 '
61110 ' Call of GSX with one INTIN parameter only
61120 '
61130 contrl%(2)=0 : contrl%(4)=1 : GOTO 61000

```

Part II

REFERENCE

CONTENTS

1. Introduction	127
1.1 Metalanguage	128
2. The Elements of BASIC	129
2.1 Character Set	129
2.2 White Space	129
2.3 Names	129
2.4 Numbers	130
2.5 String Constants	131
2.6 Data Types	131
2.7 Variables	132
2.8 Arrays	133
2.9 Type Compatability and Conversion	134
2.10 Unsigned Integer	135
2.11 Expressions	135
2.12 Functions	139
2.13 Decimal Fractions and Binary Floating Point	140
3. Direct and Program Mode	141
3.1 Direct Mode	141
3.2 Program Mode	142
3.3 Suspending BASIC	142
3.4 Run Only – Special Direct Mode	142
4. The Line Editor and Simple Line Input	145
4.1 Simple Line Input	146
4.2 The Screen Line Editor	147

5.	Overview of Commands and Functions	151
5.1	Creation of Programs	151
5.2	Loading and Running of Programs	152
5.3	Program Termination	152
5.4	Miscellaneous	153
5.5	Control Structures	153
5.6	Variables	154
5.7	Console I/O	154
5.8	Line Printer Output	155
5.9	Files	155
5.10	Constant Data	158
5.11	Arithmetic Functions	158
5.12	String Functions	159
5.13	Type Conversion Functions	159
5.14	Machine Level Operations	160
5.15	Error Trapping	160
5.16	Program Development	161
6.	Introduction to Printing	163
6.1	Operation of a Print Statement	164
6.2	Control Character Handling	164
6.3	Logical Position	164
6.4	Free Format Printing	165
6.5	USING Formatted Printing	166
7.	Control of output devices	169
7.1	Specification of control sequences	170
7.2	Transmission of control sequences	170
7.3	Interpretation of control sequences	170

8.	Introduction to File Handling	173
8.1	File Names	174
8.2	Disc and Directory Operations	174
8.3	Opening and Closing Files and File Numbers	176
8.4	Sequential File Handling	177
8.5	Random File Handling	178
9.	Keyed File Handling and Multiuser file handling	181
9.1	Consistency of the Data and Index Files	182
9.2	Multiuser Systems - Record and File Locking	182
9.3	Finding Records and Current Position in a Keyed File	186
9.4	Overview of Facilities and Recommended Use	187
9.5	General Remarks on Jetsam Commands and Functions	192
10.	The Commands and Built In Functions	195
10.1	Definition of Common Terms	195
10.2	Definition of Common Terms (Jetsam and Multiuser)	197
10.3	Common terms for MSDOS2	198
10.4	A-Z of Commands and Built in Functions	199 – 388
	Appendix I:Initialising BASIC	389
	Appendix II: Error Numbers and Messages	391
II.1	Ordinary BASIC Errors	391
II.2	Disc and File Related Errors	394
II.3	Jetsam Specific Error Numbers	396

Appendix III: External Routines	397
III.1 Data Formats	397
III.2 USR Functions	398
III.3 CALLEd Subroutines	400
III.4 Mallard-86 and Segment Registers	402
Appendix IV: BASIC Keywords	403
Appendix V: Hints on the Use of Jetsam and Multiuser Facilities	405
V.1 Numeric Keys	405
V.2 Sequential processing of Keyed Files in Multiuser Environment	406
V.3 Sub-Keys	406
V.4 Straightforward Multiuser Random File Handling	407
V.5 Guard Record	407
Appendix VI: Installation of BASIC	409
VI.1 Screen requirements	410
VI.2 Keyboard Commands for Editor	410
VI.3 Install Program	411
VI.4 Running Install	411
Appendix VII: The Command editor	415
Appendix VIII: ASCII Character Set	421
Appendix IX: Trigonometrical Functions	423
Appendix X: Binary Floating Point	425
X.1 Brief Introduction	426
X.2 Decimal Fractions & BFP	429
X.3 BFP & Financial Calculations	431
X.4 BFP & Scientific/Engineering Calculations	432

Chapter 1

Introduction

The BASIC interpreter is capable of executing the commands given in Chapter 10. Each command is identified by one or more leading keywords, and may have a number of parameters. In general each parameter may be an expression, involving constants, variables and functions. Strings and various forms of numeric data types are supported, as are sequential and random file handling.

BASIC includes a general purpose keyed file manager, called Jetsam. This can create, access and modify keyed files. Facilities exist to provide indexed and sequential access by key value to data records.

Where the operating system will support it the multiuser version of Jetsam may be used. This provides file and record locking facilities for ordinary sequential and random files as well as for Jetsam keyed files.

Commands are presented to BASIC on lines. A line may comprise several commands, separated by colons, limited only by the line length. In Direct Mode lines are input from the console. In Program Mode lines are read from the current program in memory.

In Direct Mode it is possible to add and remove lines from the program, and to amend existing lines. BASIC includes a line editor which allows the current line to be amended while it is being entered and before it is obeyed in Direct Mode.

A 'Run-Only' subset of BASIC is supported, for use with program packages written in BASIC. There are performance advantages to using a Run Only version once a program is no longer under development. The Run Only versions exclude the following features:

- Direct Mode, and hence Direct Mode commands such as EDIT
- The line editor
- Control-A (Amend previous line)
- Control-C (Break in)
- Control-S (Suspend BASIC).

Where a command is described as returning to Direct Mode, Run Only versions leave BASIC and return to the system (except in some cases where the Special Direct Mode is entered – see Section 3.4).

1.1 Metalanguage

In order to describe commands and their parameters a simple metalanguage is used. The form of each command is described as it appears when entered, with any variable or optional parts shown by 'place-holders' which refer to a concept defined elsewhere.

A concept is represented by its name written in *italic (slanted) text*. For example, in various places an expression yielding a numeric value is required: this is represented by:

numeric-expression

Anything shown in computer style typeface is required as given. For example the STOP command takes the form:

STOP

Where there is an optional part in a definition, the optional part is enclosed in square brackets. For example, if a numeric expression were to be optional then it would appear:

[numeric-expression]

If an optional part may be repeated (so may appear any number of times, including none at all) an asterisk is appended after the closing square bracket. For example, a string of digits, requiring at least one digit, would appear:

*digit[digit]**

In many places a list of objects separated by commas is used. Such lists are expressed in a special form, which is best illustrated by example, thus:

list-of: expression *is: expression[, expression]**

or: list-of: [#] number *is: [#] number[, [#] number] **

Note that the list may be a single object. If the list contains more than one object, then each additional object must be preceded by a comma.

Chapter 2 contains the definitions for the fundamental elements of BASIC. At the beginning of Chapter 10 a number of common concepts are defined, many in terms of the fundamental elements.

The Elements of BASIC

2.1 Character Set

BASIC assumes that the ASCII character set is in use. Character values less than space (value 32) are treated as non-printing, and several have special meaning to BASIC.

Where a different character set is in use, the characters having special meaning may differ from those shown in this manual. Details are given in Appendix VIII.

When processing command lines BASIC does not distinguish between upper and lower case, except in strings.

2.2 White-Space

Spaces, Tabs and Line Feeds (or any combination thereof) count as *white-space*. *White-space* is not significant in command lines, except where it has the effect of terminating an item or in a string. Note that a Carriage Return immediately following a Line Feed is also ignored.

BASIC will perform space compression, in which all *white-space* is removed from lines when they are added to the current program. BASIC generates a space to separate variable names and keywords when required. Space compression reduces a program's store requirements.

2.3 Names

Variables and Keywords have *names*. The first character of a *name* must be alphabetic, further characters may be alphanumeric or dot. Names may be at most forty characters long, all of which are significant. A *name* is terminated by any character which may not be part of a *name*.

Keywords may include other characters, and some are expressed as more than one word.

Unlike some dialects of BASIC this dialect requires that variable names are separated from keywords by at least one "non-name" character. This means that

white-space must sometimes be used as a separator. If the separator were not required it would restrict variable names to exclude any which included keywords as part of the name.

2.4 Numbers

A *number* may be any of the following:

unscaled-number[*type-marker*]

scaled-number[*type-marker*]

based-number

line-number

See Section 2.6 below for a discussion of *type-markers*.

An *unscaled-number* may take any of the following forms:

digits

digits[*digits*]

[*digits*].*digits*

where *digits* is one or more decimal digits. *White-space* is allowed in *unscaled-numbers*, and may be used at will as an aid to number input.

A *scaled-number* may take either of the following forms:

*unscaled-number*E[*sign*]*digits*

*unscaled-number*D[*sign*]*digits*

where *sign* is plus or minus, and *digits* is one or more decimal digits. *White-space* is allowed around the non-digit characters of the exponent part, but not within the *digits* portion. The value of the number is the value of the *unscaled-number* multiplied by the power of 10 given by the *sign* & *digit*, the exponent part. (Note that the absolute value of the exponent part may not exceed 99.) The difference between the E and the D form of the exponent part is described below.

A *based-number* may take any of the following forms:

&*octal-digits*

&*Octal-digits*

&*Hhexadecimal-digits*

where *octal-digits* are digits in the range 0...7, *hexadecimal-digits* are digits in the range 0...9 and letters in the range A...F (or a...f). The *based-number's* decimal equivalent may not be greater than 65535, which is mapped to the two's complement Integer range -32768 to 32767. *White-space* is allowed around the non-digit characters, but not within the *digits* portion.

A *line-number* takes the form:

digits

where *digits* is at least one decimal digit. Line numbers are limited to the range 0...65534.

numbers are terminated by any character which may not appear in a *number*.

2.5 String Constants

A *string-constant* is an arbitrary collection of characters enclosed in double quotes, or starting with double quotes and terminated by end of line – in which case any trailing *white-space* is not included in the string. Characters outside the normal ASCII printing range may appear in string constants, with the exception of Null (character value 0).

2.6 Data Types

BASIC handles two broad types of data, numeric and string.

Strings may be from 0 to 255 characters long. The characters in a string are represented by values in the range 0...255 (ie. they are bytes). It is assumed that the ASCII character set is in use, but most string processing is independent of the values of the component characters.

Numeric data may take one of three forms – Integer, Single Length and Double Length:

Integer data is held as two byte two's complement numbers, so can take values in the range -32768 to +32767. In some instances the sixteen bit two's complement form is interpreted as an unsigned value in the range 0...65535 (see Section 2.10).

Single Length data is held in a four byte binary floating point format, with three bytes of mantissa and one byte of binary exponent. The largest absolute value representable is approximately $1.7\text{E}+38$, the smallest (other than zero) is approximately $2.9\text{E}-39$. The three byte mantissa affords a little over seven decimal digits of precision.

Double Length data is held in an eight byte binary floating point format, with seven bytes of mantissa and one byte of binary exponent. The largest absolute value that can be represented is approximately $1.7\text{E}+38$, the smallest (other than zero) is approximately $2.9\text{E}-39$. The seven byte mantissa affords a little over sixteen decimal digits of precision.

The type of a *number* is implied by its form:

Based-numbers are always Integer, the unsigned value of which is mapped to its two's complement equivalent (unsigned values greater than 32767 map to *unsigned-value* – 65536).

Unscaled-numbers are treated as Integer if there is no decimal point and the value can be represented in the Integer range. Otherwise the number is treated as Single Length if there are seven or fewer significant digits and Double Length if there are eight or more significant digits.

Scaled-numbers are Single or Double Length, depending on the number of significant digits or the letter which introduces the exponent part. E implies Single Length, unless the unscaled part has eight or more significant digits, and is therefore already Double Length. D implies Double Length.

The type of an *unscaled-number* or of a *scaled-number* may be changed from its implicit type by a trailing *type-marker*. The type markers are:

% which forces the number to be rounded to the nearest Integer. If the value is beyond the range of an Integer this causes an overflow error.

! which forces the number to Single Length format.

which forces the number to Double Length format.

(Note that these are character set dependent (see Appendix VIII) and that they are the same as the type markers on variable names, see below.)

Occasionally BASIC requires an integer value in the range 0...65535 held in a two byte Integer form. It does this by interpreting the usual two's complement Integer form as its unsigned equivalent – where signed values less than zero are interpreted as 65536+*value*. See Section 2.10.

2.7 Variables

Variables have three attributes – name, type and organisation. A *variable-name* takes the form *name[type-marker]*. The type of data referred to by a *variable-name* is either given by the *type-marker*, or is assumed to be the current default. The *type-markers* are:

% for Integer

! for Single Length

for Double Length

\$ for String

(Note: Character set dependent – see Appendix VIII.)

The default type assumed depends on the first character of the *variable-name*, and may be set dynamically by DEFINT, DEFSGL, DEFDBL and DEFSTR commands. The initial default for all variables is Single Length.

Variables may be *simple-variables* which have a single value, or *array-variables* which are a collection of values of the same type. Arrays are described below.

Note that the same *name* may be used for variables of different types and organisations, and that these constitute different variables.

Variables do not need to be declared in BASIC. They are brought into being by their first use, with a value of zero or null string as appropriate.

2.8 Arrays

BASIC supports arrays of all data types. An array is a collection of data items of the same type, each of which is referenced by the array variable name followed by a suitable number of subscripts, thus:

variable-name (*subscripts*)

where *subscripts* is: *list-of: integer-expression*

and the *integer-expression* yields a value in the correct range.

(Note that square brackets may be used instead of round brackets.)

The type of the data in the array is given by the type of the *variable-name*.

An array may have any number of dimensions. Arrays may be declared explicitly in a DIM command, or implicitly by use. When an array is declared the number of dimensions and the upper bound on each is set. (In an implicit declaration the upper bound taken is 10.) Thereafter any reference to the array must use the same number of subscripts, and each one must be in range. It is not possible to change the dimensions of an array.

The lower bound on all array subscripts is 0 by default, or may be set to 0 or 1 by the OPTION BASE command. It is not possible to change this lower bound while any arrays exist.

2.9 Type Compatibility and Conversion

BASIC generally treats all numeric types as being compatible with each other. String type is only compatible with itself.

When dealing with numbers of different types BASIC will type convert automatically. Where there are no other constraints numbers are "widened" to the "largest" type involved, so Integers are widened to Single Length, which are in turn widened to Double Length. Widening can never fail.

Numbers may also be forced to a given representation. Double Length to Single Length is straightforward – the mantissa is rounded from seven bytes to three bytes: while this involves a considerable loss of precision, it seldom causes Overflow. Converting either floating point form to Integer requires the number to be rounded to an integer, which must then lie in the Integer range of -32768 to $+32767$, otherwise an Error 6 (Overflow) is generated. Converting any form to an Unsigned Integer is described in Section 2.10 below.

2.9.1 Rounding

There are several different ways in which a number in one form may be rounded to another, smaller, form. If a given number X in the larger form cannot be exactly represented in the smaller, it will be bracketed most closely by two values $X1$ and $X2$ (which can be exactly represented), where $X1 < X < X2$. The common rounding schemes choose one of these values thus:

- **Truncation or Round Toward Zero**
The excess digits are simply discarded; positive $X \rightarrow X1$, negative $X \rightarrow X2$
- **Round Away from Zero**
Positive $X \rightarrow X2$, Negative $X \rightarrow X1$
- **Round Toward -Infinity**
 $X \rightarrow X1$
- **Round Toward +Infinity**
 $X \rightarrow X2$
- **Round to Nearest.**
 $X \rightarrow X1$ or $X2$, depending on which is nearest to X

Various schemes exist for choosing between $X1$ and $X2$ when X is exactly half way between the two. BASIC rounds away from zero in this case.

Unless otherwise stated, the term "rounding" refers to Round to Nearest as defined above.

2.10 Unsigned Integer

Occasionally BASIC requires an integer value in the range 0...65535 held in a two byte Integer form – for example PEEK requires a machine memory address.

Integer (two byte two's complement) values are interpreted as their unsigned equivalent – with signed values less than zero interpreted as 65536+*value*.

If BASIC is given a Floating Point value when Unsigned Integer is required, it is first rounded to an integer value (but not yet to a two byte Integer), which must be in the range -32768 to 65535. This integer value is then converted to two's complement Integer form, mapping all values greater than 32767 to *value*-65536. The newly formed Integer is then interpreted as unsigned as described above. (Allowing negative values to map to Unsigned Integer is, at first sight, a little curious – it is done to ensure that Floating Point values behave in the same way as Integers.)

2.11 Expressions

There are four classes of expression – numeric, string, relational and logical. The precedence of operators is designed so that expressions work as might be expected. Where the order of evaluation is not forced either by the precedence rules or by brackets, evaluation proceeds from left to right.

2.11.1 Numeric Expressions

The syntax of numeric expressions is:

<i>numeric-expression</i>	is:	<i>numeric-item</i> [<i>operator numeric-expression</i>]
<i>numeric-item</i>	is:	[<i>unary-operator</i>] <i>numeric-item</i>
	or:	<i>numeric-constant</i>
	or:	<i>numeric-variable</i>
	or:	<i>numeric-function</i>
	or:	(<i>numeric-expression</i>)
	or:	(<i>relational-expression</i>)

where *numeric-variable* is a reference to a simple variable or an array item of any of the numeric types. A *numeric-function* is a function returning any of the numeric types.

Chapter 2: The Elements of BASIC

The *operators* and *unary-operators*, in order of precedence, are:

^	Exponentiation	Raises the value on the left to the power of the value on the right. Forces both values to Single Length before operation, and produces a Single Length result. (Note: The character used as the exponentiation operator is character set dependent; see Appendix VIII)
+	Unary Plus	Note that both <i>unary-operators</i> have lower precedence than Exponentiation.
	Unary Minus	
*	Multiply	Floating point or Integer multiplication. Floating point division. If either value is Integer it is forced to Single Length before operation.
	/	
\	Integer Division	Forces both values to Integer. The result is truncated to an Integer. (Note: The character used as the operator is character set dependent; see Appendix VIII)
MOD	Integer Modulus	Forces both values to Integer. The result is the remainder after Integer division.
+	Addition	Floating point or Integer addition. Floating point or Integer subtraction.
	Subtraction	

Note that:

- i. Exponentiation binds tightest.
- ii. Unary Plus and Unary Minus have equal precedence.
- iii. Multiply and Divide have equal precedence.
- iv. Integer Division and Modulus have equal precedence, and that their precedence is lower than standard Multiply and Divide. This implies that:
 $x \setminus y * y = x \setminus (y * y)$ and $x \setminus y * y \neq (x \setminus y) * y$
- v. Integer Modulus is defined so that:
$$x \text{ MOD } y = x - ((x \setminus y) * y)$$

so the sign of $x \text{ MOD } y$ is always the same as the sign of x (unless the result is zero).
- vi. Addition and Subtraction have equal precedence.
- vii. Except where the operator has particular requirements, the values on either side are forced to the same type by widening.

Division by zero generates an Error 11. If there is an ON ERROR GOTO set, then it is invoked in the usual way. If there is no ON ERROR GOTO set then:

- the "Division by zero" error message is generated.
- Integer division (\) by zero causes execution to cease.
- Floating point division (/) by zero does not terminate execution, because the zero may be the result of underflow in a previous operation. Instead the largest representable number (approx. 1.7E+38) with appropriate sign is returned, and execution continues.

In the event of overflow when the arguments are Integer, the values are widened to Single Length and the operation repeated.

In the event of overflow when the arguments are floating point an Error 6 ("Overflow") is generated. If there is an ON ERROR GOTO set, then it is invoked in the usual way. If there is no ON ERROR GOTO set, the "Overflow" error message is generated, the largest representable number (approx. 1.7E+38) with appropriate sign is returned, and execution continues.

2.11.2 String Expressions

The syntax of string expressions is:

<i>string-expression</i>	is: <i>string-item</i> [+ <i>string-expression</i>]
<i>string-item</i>	is: <i>string-variable</i>
	or: <i>string-constant</i>
	or: <i>string-function</i>
	or: (<i>string-expression</i>)

where *string-variable* is a reference to a simple variable or an array item of type string. A *string-function* is a function returning a value of type string.

The effect of the + operator on strings is to concatenate them. A new string is produced which is the first string immediately followed by the second. If the resulting string would be longer than 255 characters, an Error 15 is generated.

2.11.3 Relational Expressions

Relational expressions compare two numeric values or two string values. Thus:

<i>relational-expression</i>	is: <i>numeric-expression</i> <i>relation-operator</i> <i>numeric-expression</i>
	or: <i>string-expression</i> <i>relation-operator</i> <i>string-expression</i>

Note that *relation-operators* have a lower precedence than any of the operators in either string or numeric expressions, and a higher precedence than any logical operator.

The *relation-operators* are :

<	Less than
<= =<	Less than or Equal
=	Equal
>= =>	Greater than or equal
>	Greater than
<>	Not equal

Where the arguments are numeric the types are made the same by the process of widening described in Section 2.9.1 above. The meaning of the relation is as might be expected.

Where the arguments are strings, the meaning of the relations requires some explanation:

- Two strings are equal when they are the same length, and corresponding characters are the same.
- One string is less than another if:
 - either they are equal up to the end of the first string and the second string is longer
 - or the first character which is not the same in the two strings is smaller in the first string (where the values representing the characters are being compared)

BASIC does not support a Boolean type. Relational expressions yield an Integer value, -1 for True and 0 for False. Note that the IF and WHILE commands treat a value of 0 as False and any other value as True.

2.11.4 Logical Expressions

BASIC does not support a Boolean type. Logical expressions perform bit-wise boolean operations on Integers. Logical expressions are :

argument[*logical-operator argument*]

where *argument* is: NOT *argument*
or: *numeric-expression*
or: *relational-expression*
or: (*logical-expression*)

The two arguments for a logical operator are forced to Integer; Error 6 results if an argument will not fit into the Integer range. The operation is then performed for each bit of the 16 bit two's complement representation of the Integer.

The NOT unary operator inverts each bit in the argument (0 becomes 1, and vice versa).

The dyadic operators, in order of precedence, and their effect on each bit are:

AND	Result is 0 unless both argument bits are 1
OR	Result is 1 unless both argument bits are 0
XOR	Result is 1 unless both argument bits are the same
EQV	Result is 0 unless both argument bits are the same
IMP	Result is 1 unless the right argument bit is 1 and the left 0

The result of a relational expression is either -1 or 0. The representation for -1 is all bits of the Integer = 1; for 0 all bits of the Integer are 0. The result of a logical operation on two such arguments will yield either -1 or 0.

2.12 Functions

Functions are subroutines which take a number of parameters, and which return a single value. Functions are invoked by their use as arguments in expressions, the value of the argument being the value returned by the function. There are three classes of function, built-in, user and external user.

2.12.1 Built-in Functions

These are part of the BASIC language, and are described in Chapter 10. Unlike user functions the type of the result is not necessarily dictated by the type of the function name. For example the function ABS returns a result of the same type as its argument. Some built-in functions have optional parameters, and some allow different parameter types, neither of which is possible with user functions.

2.12.2 User Functions.

User functions are defined by the DEF FN command. This associates a name with a formal parameter list and an expression.

Function names take the form `FNname[type-marker]`, where the FN serves only to distinguish the name from variable names, and does not count toward the forty possible characters of the *name* part. As with variables, functions with the same name but different types are different functions.

The formal parameter list declares a number of variables local to the function. When the function is invoked, actual parameters must be given which conform in number and type to the formal parameters – noting that BASIC will automatically

change numeric types as required. The actual parameters are expressions, which are evaluated before the function is called, and whose values are copied to the formal parameters.

The body of the function is a single expression. It is not possible to have any form of control structure in a function. The expression may include references to other functions and to global variables as well as the formal parameters – though, of course, a formal parameter with the same name as a global variable makes the global variable's value inaccessible.

When a function is evaluated the result of the expression is converted to the same type as the function name, and the converted value returned.

2.12.3 External User Functions

An external user function is a machine code subroutine whose address in memory has been declared in a DEF USR command. External user functions take one parameter of any type, and may return a value of any type. See Appendix III for a full description of the support for externally developed subroutines.

2.13 Decimal Fractions and Binary Floating Point

It is a sad fact that few decimal fractions have an exact representation as binary floating point numbers. Floating point numbers are rounded before they are printed so the inexact representation may be obscured.

The effect may be seen when comparing values which are the result of separate sequences of arithmetic operations. The two values may appear equal when printed, but their binary representations are different. The same effect may be seen if two such numbers are subtracted and the result is not zero.

The effect may also be seen when Single Length numbers are converted to Double Length. For example if 0.1! is converted to Double Length the result is 0.1000000014901161#, which is close, but not exactly the same.

Unless decimal fractions are avoided altogether, by multiplying all numbers by suitable powers of ten (treating all money values as pence, for instance) it is impossible to avoid these effects. They can be reduced by rounding explicitly using the built-in function ROUND, which will round numbers which print the same to exactly the same value. Alternatively the slight differences may be retained, and account taken of their existence when numbers are compared or subtracted.

Chapter 3

Direct and Program Mode

BASIC works in one of two major modes, Direct and Program. In Direct Mode BASIC takes lines typed at the console and processes them immediately the terminating carriage return is entered. In Program Mode BASIC takes lines from the program currently in memory and executes the commands in each. With very few exceptions all commands may be used in either mode.

Note that Direct Mode is not available in Run Only versions of BASIC, but see Section 3.4 below.

3.1 Direct Mode

Direct Mode may be sub-divided into:

Direct Commands

In which commands are entered which are then immediately executed

Program Input

In which new or replacement program lines are entered, or existing program lines are removed

Program Amendment

In which existing program lines are altered

Direct Commands mode is straightforward. It is possible to use variables in Direct Commands mode. All variables are lost, however, when Program Mode is entered using the RUN command.

Program Input may be done simply by entering a line starting with a number. When the line is terminated (by carriage return) it is inserted into the current program, replacing any existing line with the same number. (Note that a line consisting solely of a line number deletes any existing line with that number.) The AUTO command causes BASIC to generate a sequence of line numbers for Program Input.

Program Amendment is supported by the EDIT command, see Chapter 4.

Note that BASIC does not perform any checking of program lines when they are entered or amended.

3.2 Program Mode

In Program Mode BASIC automatically steps through the lines of the current program, obeying the commands it finds there. Lines are taken in line number order, except where explicit control structures dictate otherwise.

Error Processing Mode is a sub-mode of Program Mode, and is entered when an error occurs while an ON ERROR GOTO is active.

Except in Run Only versions, Control-C from the console while a program is running will interrupt it, causing BASIC to return to Direct Mode with a Break message. Provided the program is not altered in any way, it may be restarted by CONT as if the Break had never happened.

Variables are preserved when BASIC returns from Program to Direct Mode, which may be useful when debugging a program.

3.3 Suspending BASIC

Control-S from the console causes BASIC to suspend program execution, or suspend output to the console or printer. A second Control-S allows BASIC to continue, as if nothing had happened. Any other character after Control-S allows BASIC to continue, but the character is remembered and may be processed later.

Like Control-C, Control-S has no effect in Run Only versions.

3.4 Run Only – Special Direct Mode

There are two variants of Run Only versions – one with, and one without, a Special Direct Mode.

The variant without the Special Direct Mode returns immediately to the system level if the current program fails or otherwise comes to a halt.

The variant with the Special Direct Mode enters it when an error is detected or when a STOP or END command is executed. In this mode BASIC produces the prompt 'Direct Mode' and will accept a limited number of commands, namely:

DIR	MID\$	GOTO
PRINT	LSET	GOSUB
LET	RSET	SYSTEM

BASIC will not accept any other commands, nor will it allow new program lines to be entered. Assignments must be preceded by LET.

This mode is provided to allow the state of a failed program to be examined (PRINT) and perhaps altered (LET, MID\$, LSET, RSET). If it is possible to 'patch up' the program it may be restarted (GOTO or GOSUB). If the program cannot be restarted it can be abandoned (SYSTEM).

If this mode is used, it is expected that the user will either be an expert on the failed program, or be advised by an expert. In the event of a program failing, it may be possible to restart it in order to avoid the loss of data.

The Line Editor and Simple Line Input

The Line Editor is entered by using the EDIT and AUTO commands, by typing Control-A, or automatically in the event of a syntax error (Error 2). BASIC is implicitly in the line editor during line input in Direct Mode – so the current line may be edited while it is being entered.

At other times when BASIC fetches lines from the console a simple line input routine is used, which only allows character delete at the end of the current line.

The EDIT command enters the line editor ready to amend the given program line. Syntax errors detected while a program is running cause BASIC to enter the line editor as if an EDIT command had been issued for the offending line. During AUTO when the next line number is generated and that line already exists it is presented for editing. At other times during AUTO, and in Direct Mode, new lines may be edited as they are entered. Control-A typed immediately after a carriage return in Direct Mode allows the previous line to be edited and re-entered.

If the line number of a program line is changed while editing it the result will be inserted in the new position in the program, replacing any existing line, without affecting the original line.

The line editor is not available in Run Only versions. Syntax errors cause BASIC to exit back to the system level or to special direct mode – see Section 3.4.

Two line editors are supported. One is a 'screen based line editor', which displays the latest state of the line at all times and provides a 'point and change' style of editing. The other is a command driven editor which does not display the current state of the edited line.

In order to support the screen editing BASIC requires knowledge of a small number of screen cursor movement commands. To make the editor as easy to use as possible the form of the edit commands may be tailored for a particular keyboard. There is a table within BASIC which defines the form of the required screen commands and of edit commands. Setting up this table is known as 'installing BASIC'.

Copies of Mallard BASIC for the IBM PC (or compatible) or for one of the Amstrad computers are supplied with the screen-based editor installed for these machines. Copies for use on other machines are supplied with an installation program. Appendix VI describes the use of the installation program. The installation program is not supplied with versions of BASIC which are supplied installed for a particular computer.

The advantage of the screen editing is that it is straightforward to use. The disadvantage is that an installed copy of BASIC is machine specific, and the editor may not function on other machines. The command editor, on the other hand, requires no installation, and is therefore machine independent. The command editor is, however, less convenient to use. The use of the command editor is described in Appendix VII.

Since Run Only versions do not include either form of line editor they do not require installation. It may be useful, however, to use the installation program to set the console width and Auto CRLF column, which otherwise both default to 80.

4.1 Simple Line Input

When fetching a line for INPUT, LINE INPUT and RANDOMIZE, a simple line input routine is used. (On Run Only versions all line input is done this way.) Simple line input accepts all characters other than control characters and appends them to the current line. The following control characters have special effects:

- Control-C (&H03) abandons the input, the line is ignored. Break action is taken.
- Carriage Return (&H0D) completes the line and issues it to BASIC.
- Backspace (&H08) or Rubout (&H7F) deletes the last character in the line; Backspace (&H08), Space(&H20), Backspace (&H08) is sent to the console.
- Tab (&H09) is inserted into the line and spaces are sent to the console up to the next tab position – tabs are set at every eighth column.
- Line Feed (&H0A) is inserted into the line and Carriage Return followed by Line Feed are sent to the console.

All other control characters (in the range &H00...&H1F) are ignored. All other characters (including values in the range &H80...&HFF) are appended to the line and reflected to the console.

4.2 The Screen Line Editor

The editor is a 'screen based line editor'. That is to say it presents the current line on the screen and allows the cursor to be moved around within the line, and text to be inserted or removed at the cursor position. At all times the editor displays the latest state of the line.

The commands accepted by the line editor are described here, and referred to by name. The actual keystrokes which invoke the commands are set when BASIC is installed, and may be any combination of characters, though it is expected that they will be single control codes or a control code followed by one or two other characters.

The line being edited may be too long to fit on a single line of the screen. The term 'screen line' refers to the portion of the current line which is displayed on a single line of the screen. The editor needs to know the width of the screen and whether the screen is performing Auto CRLF. This information is installed in BASIC, but may be altered by the WIDTH command.

4.2.1 Cursor Movement

The cursor may be moved left and right along the current line, and up and down if the line occupies more than one screen line. When moving up and down the cursor may not be moved beyond the first or last screen lines used. The cursor may also be moved forward to the next occurrence of a given character.

The cursor is not constrained to positions corresponding to characters in the line, it may, for example, be moved beyond a line feed or into the 'dead' space between a tab and the next character. If the cursor is positioned on a character when text is inserted or deleted it will be moved left or right before the insertion or deletion is performed.

The movement commands are:

Cursor Left Move one position left.

If the cursor is positioned on the first character of the first screen line this command is ignored.

If the cursor is positioned at the lefthand end of the second or subsequent screen line it moves to the righthand end of the previous screen line.

Cursor Right Move one position right.

If the cursor is positioned on the last character of the last screen line this command is ignored.

If the cursor is positioned at the righthand end of the penultimate or earlier screen line it moves to the lefthand end of the next screen line.

Cursor Up Move vertically up one screen line.

If the cursor is on the first screen line this command is ignored.

Cursor Down Move vertically down one screen line.

If the cursor is on the last screen line this command is ignored.

Find Character Move forward to the next occurrence of the given character.

The command takes the next character typed (which may be any character) and searches forward down the line for it. Before the search starts the cursor is moved one position to the right so that the character at the initial position is not considered. If no such character is found the cursor is positioned at the end of the line.

4.2.2 Adding Text

All characters typed which do not form editor commands are added to the current line – except that control characters (values &H7F and in the range &H00...&H1F) other than Tab and Line Feed, are rejected. The way in which the character is added to the line depends on the Insert/Overwrite mode. The editor always starts in Insert mode each time a line is edited.

The commands associated with the addition of text are:

Insert/Overstrike toggle If in insert mode then set overstrike mode and vice versa.

When a character is entered into the line in insert mode all characters to the right of the current position move right to make room.

When a character is entered into the line in overstrike mode the current character is deleted and replaced. Note that overstriking tabs and line feeds causes the line layout to change.

Insert mode is always selected when editing starts.

Literal Character: '\'. The character immediately following the Literal Character command is not treated as a possible command, but as a character for addition to the current line. The character may be Tab (&H09), Line Feed (&H0A) or any character whose value is in the range &H20...&HFF except &H7F.

The main use of the literal character command is in adding Tabs or Line Feeds when these characters are used as editor commands.

The character '\' may be added to the line using this command, by typing '\' twice.

Tab and Line Feed control characters may appear in the line, and may be added to it. These have special effects as follows:

Tab is represented as spaces to the next tab position. Tab positions are at every eighth column. When the cursor is positioned on the first space then it is on the tab; when it is on a subsequent space then it is after the tab.

Line Feed is represented as a space followed by a new screen line. When the cursor is positioned on the space or after the space but on the same screen line then it is on the Line Feed.

4.2.3 Deleting Text

Characters may be deleted implicitly by adding to the line in Overstrike mode. There are three commands which explicitly remove text :

Forwards delete If there is no character under the cursor then move forwards until there is one. Delete the character under the cursor.

Backwards delete Move the cursor one position left. If there is no character under the cursor then move backwards until there is one. Delete the character under the cursor.

Delete to character Delete from the current cursor position up to but not including the given character.

The command takes the next character typed (which may be any character) and searches forward down the line for it, deleting characters on the way. Before the search starts the current character is deleted – so the character at the initial position is not considered in the search. If no such character is found the whole of the rest of the line will be deleted.

4.2.4 Finishing an Edit

Finishing an edit may either issue the new version of the line to BASIC or may abandon it altogether. The commands are:

Complete the edit and issue the new line If the line has a line number then it is inserted in place in the current program, possibly replacing an existing line. If in AUTO the next line number is generated and the editor is re-entered for that line. Note that if the line number has been changed during the edit the original line is not affected.

If the line does not have a line number then it is treated as a Direct Mode command line, and is executed.

(It is recommended that Carriage Return is used for this command).

Abandon editing The result of the edit is discarded. Abandoning the current edit also has the effect of terminating AUTO and returning BASIC to Direct Mode. (It is recommended that Control-C is used for this command, to be consistent with the usual BREAK command).

Overview of Commands and Functions

This chapter is not intended to describe the commands in detail, that is done in Chapter 10. Here the commands are described in groups, in order to give some idea of their purpose and to place them in context.

5.1 Creation of Programs

AUTO, DELETE, EDIT, LIST, LLIST, NEW, RENUM, SAVE

The commands in this group are all useful when preparing a new program, or amending an existing one. Run Only versions exclude all these commands other than **SAVE** and **DELETE**.

AUTO is an aid to the entry of a new section of program. It generates line numbers as the new program lines are entered.

DELETE removes lines of program en masse.

EDIT allows existing lines to be amended.

LIST and **LLIST** produce listings of all or part of a program to the console or to the line printer, respectively.

NEW clears the program area, ready for a new program to be entered.

RENUM allows all or part of a program to be renumbered. References to lines which are renumbered are updated.

SAVE allows the current program to be written to a file. The program may be written in one of three forms:

- **ASCII** a text file, in the same form as produced by **LIST**
- **Compressed** a copy of the internal form of the program, which requires less disc space and will load more quickly than ASCII files.
- **Protected** similar to **Compressed**, except that the program is encrypted. When a **Protected** program is loaded all commands which give the user access to the program text are inhibited. Once a program has been protected it cannot be unprotected.

5.2 Loading and Running of Programs

CLEAR, **CHAIN**, **CHAIN MERGE**, **COMMON**, **COMMON RESET**, **HIMEM**, **LOAD**, **MEMORY**, **MERGE**, **RUN**

CLEAR clears away all variables, closes all files and generally resets BASIC, except that the current program is retained. The memory available to BASIC, the size of stack to be used, the maximum number of files and the maximum random record size may be changed at this point.

MEMORY allows the memory available to BASIC, the size of stack to be used, the maximum number of files and the maximum random record size to be changed.

HIMEM The HIMEM function returns the address of the highest byte in memory used by BASIC.

CHAIN, **CHAIN MERGE**, **COMMON**, **COMMON RESET** **CHAIN** and **CHAIN MERGE** enable one program to load and enter another. With **CHAIN MERGE** all or part of the current program may be retained. Variables may be retained past **CHAIN** or **CHAIN MERGE** – all may be retained, or only those specified in **COMMON** statements. At any time **COMMON RESET** will dispose of all variables not specified in **COMMON** statements.

MERGE allows a program on disc to be merged into the current program. **MERGE** is intended for program development, and is not available in Run Only versions.

LOAD discards the current program and variables and resets all other status. A new program is loaded and may be entered at once.

RUN The simplest **RUN** command starts executing the current program. Other forms of the command cause execution to start at a given line number or are similar to **LOAD**.

5.3 Program Termination

END, **SYSTEM**

END Programs stop and return to Direct Mode after the last line has been executed. **END** may be used to terminate the program at some other point, or points.

SYSTEM The **SYSTEM** command causes BASIC to stop executing, close all files, and return to the System level.

5.4 Miscellaneous

OPTION RUN, **OPTION STOP**, **VERSION**

OPTION RUN disables the actions of **Control-C** and **Control-S**, which normally (except in Run Only versions) stop or suspend the current program.

OPTION STOP enables these actions.

VERSION is a function which returns information about the version of BASIC, about the operating system and so on.

5.5 Control Structures

FOR, **GOSUB**, **GOTO**, **IF**, **ON x GOSUB**, **ON x GOTO**, **RETURN**, **WHILE**

FOR, **WHILE** BASIC supports two forms of loop, **FOR** and **WHILE**. A **FOR** loop has a control variable associated with it, which steps through a range of values – one per iteration – until a given end value is reached. The size of each step may be specified, including negative steps. The BASIC **FOR** loop is skipped if the initial value of the control variable exceeds the end value. **NEXT** marks the end of a **FOR** loop. A **WHILE** loop repeats until the given condition becomes false. The BASIC **WHILE** loop is skipped if the condition is false the first time. **WEND** marks the end of a **WHILE** loop. **FOR** and **WHILE** loops may be nested in any combination.

IF commands may be used to choose alternative or optional actions. **IF** is followed by an expression. If the expression yields a non-zero value the **THEN** part is chosen, otherwise if it is present the **ELSE** part is chosen. Both the **THEN** and **ELSE** parts may contain more than one command, but they must all be on the same line with the **IF** command. **IF** commands may be nested on the same line.

GOSUB, **RETURN** Simple subroutines are supported. **GOSUB** calls the subroutine that starts at the line given. **RETURN** terminates the subroutine, and returns control to the next command after the matching **GOSUB**. All variables in BASIC are global. Subroutines may recurse.

GOTO causes an unconditional branch to the line given.

ON x GOSUB and **ON x GOTO** commands evaluate the expression **x**, and use the result to choose one out of a list of lines to call or jump to.

5.6 Variables

DEFINT, DEFSNG, DEFDBL, DEFSTR, DIM, ERASE, OPTION BASE

DEFINT, DEFSNG, DEFDBL, DEFSTR Where a variable or function name is given without an explicit type marker BASIC assumes a type. The type assumed depends on the first character of the name (which is an alphabetic character). **DEFINT, DEFSNG, DEFDBL** and **DEFSTR** commands may be used to define the default type for each letter. Note that these default settings may be changed at will, but that the potential for confusion is enormous.

DIM, OPTION BASE Unless otherwise explicitly declared in a **DIM** command the maximum for each subscript of an array is set to 10 by the first use of the array (implicit declaration). The minimum for each subscript of an array may be set by an **OPTION BASE** command to 0 or 1, but defaults to 0 otherwise. It is an error to attempt to change this base value once it has been set, or once an array has been declared (explicitly or implicitly).

ERASE Arrays may occupy large areas of memory. The **ERASE** command may be used to reclaim the memory used by arrays which are no longer in use.

5.7 Console I/O

INKEY\$, INPUT, INPUT\$, LINE INPUT, POS, PRINT, WIDTH, WRITE, ZONE, OPTION PRINT, OPTION INPUT, OPTION NOT TAB, OPTION TAB

INKEY\$ The **INKEY\$** function fetches a single character from the keyboard, if one is available. **INKEY\$** does not reflect the character to the console, and may be used in special purpose input routines.

INPUT fetches a line from the console, and may interpret parts of it as numeric input and parts as strings, depending on the parameters of the **INPUT** command.

INPUT\$ is similar to **INKEY\$**, except that a given number of characters are fetched from the keyboard, and **INPUT\$** waits until that number have been typed.

LINE INPUT fetches a line from the console without interpretation (but with line imaging) and assigns it to a string variable.

WIDTH All output to the console interacts with the console width, as set by a **WIDTH** command (the default width is 80). BASIC will start a new line in order to keep the console position within the console width. The current position across the console is returned by the **POS** function. If the console itself starts new lines after

a given column position, **WIDTH** may be used to inform BASIC so that unnecessary new lines can be avoided.

PRINT, ZONE **PRINT** outputs numbers and strings to the console. Numbers are output 'free format'. **PRINT** divides the console into 'print zones' which impose some order on the output. The **ZONE** command may be used to change the size of print zones. The keyword **USING** introduces a format template, which controls the output of all further numbers and strings, without reference to print zones. See Chapter 6 for a discussion of **PRINT** and related commands.

WRITE outputs numbers and strings to the console, inserting commas between each item and enclosing strings in double quotes. Numbers are output in 'free format'.

OPTION PRINT, OPTION INPUT **OPTION PRINT** allows the user to specify the address of a machine code subroutine which BASIC will use to send characters to the console rather than call the operating system. **OPTION INPUT** allows the user to specify machine code subroutines which BASIC will use to read from the keyboard.

OPTION TAB, OPTION NOT TAB allow the user to turn on and off BASIC's standard conversion of Tab (&H09) characters into spaces when they are sent to the console or printer.

5.8 Line Printer Output

LPOS, LPRINT, WIDTH LPRINT, OPTION LPRINT

These commands are equivalent to the console output ones.

5.9 Files

BASIC supports the reading and writing of disc files, with sequential or random access. In order to access a particular file it must be opened, at which time the access method is declared, and the disc file associated with a *file-number*. Thereafter the file is referenced using the *file-number*. A given *file-number* may refer to only one file at a time, but may be re-used once the previous file has been closed.

Unless otherwise set when BASIC is initialised (see Appendix I) or in **CLEAR** or **MEMORY** commands, *file-numbers* are in the range 1...3. That is to say BASIC can handle up to three files at once.

Chapter 7 contains further description of BASIC's file handling facilities.

5.9.1 Directory Access – all versions

DEL, DIR, ERA, FILES, KILL, NAME, REN, FINDS, OPTION FILES

DIR, FILES The DIR and FILES commands generate directory listings to the console.

DEL, ERA, KILL The DEL, ERA and KILL commands delete files from disc.

NAME, REN The NAME and REN commands rename files on disc.

FIND\$ The FIND\$ function may be used to find the full name of a given file, including any file attributes. The file name given to FIND\$ may include 'wild card' characters and there is provision for finding the *n*th matching name.

OPTION FILES may be used to set the default drive and user.

5.9.2 Directory Access – MS-DOS 2 version

CD, CHDIR, CHDIR\$, FINDDIR\$, MD, MKDIR, RD, RMDIR

CD and CHDIR change the current directory on a specified drive or on the default drive.

CHDIR\$ returns a string containing the current directory on the specified drive.

FINDDIR\$ looks for a given directory and returns its name if it is found.

MD and MKDIR make a new directory on a specified drive or on the default drive.

RD and RMDIR remove a directory from a specified drive.

5.9.3 File Inspection

DISPLAY, TYPE

DISPLAY and TYPE The DISPLAY and TYPE commands print a given file on the console.

5.9.4 In General

CLOSE, OPEN, RESET

OPEN The OPEN command specifies the name of the file, the access method required, and the file number which is to be used. The file name is given as a string, whose format must conform to the underlying operating system's rules.

CLOSE is the inverse operation to **OPEN**. If the file is open for Sequential Output any data still in buffers is written away. The disc file is closed and the file number released for other use.

RESET The **RESET** command closes all files and resets the file system. A **RESET** command is usually required before discs are changed.

5.9.5 Input Access – Sequential

EOF, INPUT, LINE INPUT

INPUT, LINE INPUT Input Access allows ordinary text files to be read. **INPUT** and **LINE INPUT** whose first argument is *#file-number* are similar to the console input commands, except that they read lines from file.

EOF It is an error to attempt to read past the end of an input file. The **EOF** function may be used to detect the end of file condition, and avoid reading any further.

5.9.6 Output Access – Sequential

PRINT, WRITE

PRINT Output Access allows ordinary text files to be written. **PRINT** commands whose first argument is *#file-number* are similar to the console output commands, except that data is written to file. Note that files are assumed to be infinitely wide.

WRITE commands whose first argument is *#file-number* are particularly useful for generating files to be read at some later date using **INPUT**, since **INPUT** expects items to be separated by commas and the double quotes around strings avoid any ambiguity.

5.9.7 Random Access

CVD, CVI, CVS, CVUK, CVIK, FIELD, GET, LSET, MID\$, MKD\$, MKI\$, MK\$\$, MKUK\$, MKIK\$, PUT, RSET

GET, PUT **BASIC** maintains a buffer for each file open for random access. The **GET** command reads a given record from disc into the random record buffer. The **PUT** command writes the current contents of the random record buffer to the given record on disc.

FIELD Each **FIELD** command defines a template for the data in the random record buffer. Any number of such templates may be defined for a given file, allowing different record organisations in the same file.

CVD, CVI, CVS, CVUK, CVIK, MKDS, MKI\$, MKS\$, MKUK\$, MKIK\$ In order to read and write numeric data, particularly when **FIELDs** are used, a mechanism is provided to convert the numeric data types to strings, and back again. **MKD\$, MKS\$** and **MKI\$** produce strings from Double Length, Single Length and Integer data respectively. **CVD, CVS** and **CVI** reconvert such strings. **CVUK** and **CVIK** are similar to **CVI**, and **MKUK\$** and **MKIK\$** to **MKI\$**, but operate on strings suitable for use as Jetsam keys.

5.10 Constant Data

DATA, READ, RESTORE

DATA, READ All the **DATA** commands in a program, taken in line number order, make up a sort of file. This 'file' may be read using **READ** commands.

RESTORE The position within this 'file' may be reset by **RESTORE** commands.

5.11 Arithmetic Functions

ABS, ATN, COS, EXP, FIX, INT, LOG, LOG10, MAX, MIN, RANDOMIZE, RND, ROUND, SGN, SIN, SQR, TAN

ATN, COS, SIN and **TAN** are the usual trig functions (**ATN** is Arctangent), with angles expressed in radians.

EXP returns the value of *e* raised to the given power. **LOG** returns the natural log (base *e*) of its argument. **LOG10** returns the log to the base 10. **SQR** returns the square root of its argument.

RND returns the next number in a pseudo-random sequence. **RANDOMIZE** sets a new seed for the sequence.

MIN, MAX. **MIN** takes a variable number of arguments and returns the value of the smallest. **MAX** similarly returns the largest value.

FIX and **INT** return integral values. **FIX** rounds its argument toward zero. **INT** rounds its argument toward -Infinity. **ROUND** rounds its argument to a given number of decimal places, rounding to nearest. (See Section 2.9.1 for a discussion of rounding.)

ABS returns the absolute value of its argument. **SGN** returns the sign of its argument.

5.12 String Functions

OCT\$, DEC\$, HEX\$, INSTR, LEFT\$, LEN, LOWER\$, MID\$, RIGHT\$, SPACE\$, STR\$, STRING\$, STRIP\$, UPPER\$, VAL

LEFT\$, RIGHT\$ and **MID\$** functions return part of the argument string. **LEFT\$** returns a given number of characters counting from the left end of the string. **RIGHT\$** returns a number of characters counting from the right. **MID\$** returns some portion taken from a given position in the string.

MID\$ may be assigned to, so that a portion of its argument string is replaced by another.

INSTR searches one string for the first occurrence of a given substring.

LEN returns the length of its argument string.

LOWER\$, UPPER\$, STRIP\$ **LOWER\$** returns a copy of its argument, with any upper case alphabetic characters replaced by their lower case equivalents. **UPPER\$** similarly replaces lower case by upper case. **STRIP\$** returns a copy of its argument, after stripping Bit 7 (the most significant bit) from each character.

STRING\$, SPACE\$ **STRING\$** returns a string consisting of a given character repeated a given number of times. **SPACE\$** returns a string consisting of a given number of spaces.

OCT\$, DEC\$, HEX\$ and **STR\$** produce string representations of the value of their numeric arguments. **OCT\$** produces a string of octal digits, **HEX\$** a string of hexadecimal digits. **STR\$** produces a free format decimal string. **DEC\$** produces a formatted decimal string – with format controls similar to **PRINT USING**. **VAL** takes a string and, interpreting it as a number in much the same way as **INPUT**, returns its value.

5.13 Type Conversion Functions

ASC, CHR\$, CDBL, CINT, CSNG, UNT

ASC takes the first character of a string and returns its integer value.

CHR\$ takes an integer value and returns a single character string.

CINT takes a numeric value and rounds it to Integer, rounding to nearest. The result must be in the range -32768...32767.

CSNG, CDBL CSNG takes a numeric value and converts it to Single Length. CDBL takes a numeric value and converts it to Double Length.

UNT takes a numeric value and converts it to an Unsigned Integer – see Section 2.10.

5.14 Machine Level Operations

CALL, DEFUSR, DEFSEG, INP, INPW, OUT, OUTW, PEEK, POKE,USR, WAIT, WAITW, OPTION INPUT, OPTION LPRINT, OPTION PRINT

CALL, DEFUSR, USR Two forms of external subroutine are supported. CALL invokes a machine code routine at a given address, passing a number of parameters. USR invokes a machine code function at an address previously defined by a DEFUSR command. USR passes one parameter, and returns one value as the value of the function. See Appendix III for a more detailed description.

PEEK and POKE read and write bytes of machine memory.

DEFSEG is used in Mallard-86 to specify the segment part of addresses used in CALL, USR, PEEK and POKE commands. DEFSEG commands are ignored in Mallard-80.

INP, INPW, OUT, OUTW, WAIT, WAITW INP, OUT and WAIT give access to machine I/O space. INPW, OUTW and WAITW give access to machine I/O space organised as words.

OPTION INPUT, OPTION LPRINT, OPTION PRINT allow the user to specify machine language subroutines which BASIC will use for keyboard input, printer output and console output rather than the usual operating system calls.

5.15 Error Trapping

ERL, ERR, ERROR, ON ERROR GOTO, OSERR, RESUME

ERL, ERR, ERROR, ON ERROR GOTO, RESUME When BASIC detects an error its default action is to issue an error message and return to Direct Mode. This action may be overridden by an ON ERROR GOTO command, which sets the line number of a routine to process errors. When an error occurs ERL is set to the current line number, and ERR is set to the error number. The error processing routine may inspect these variables to decide whether it is capable of recovering or not. The RESUME command may be used to return to normal running.

OSERR Some operating systems return a number of miscellaneous but interesting errors to BASIC. Rather than have a different set of BASIC errors for each such operating system BASIC provides the single error number 21 and the function OSERR. When an error 21 has occurred the function OSERR will return an operating system dependent number which will identify the error.

5.16 Program Development

CONT, STOP, TRON, TROFF

Run Only versions exclude all these commands other than STOP, which has the same effect as END.

STOP, CONT The STOP command halts a program in such a way that it may be restarted by CONT (provided the program is not altered in the meantime). STOP commands may be inserted in a program to halt it at interesting points so that the state of the program and variables may be examined.

TRON, TROFF TRON and TROFF turn on and off a tracing mechanism. When tracing is enabled, before each line is executed its number is printed on the console, enclosed in square brackets. This provides a simple trace of the program's execution.

Introduction to Printing

This chapter describes the facilities BASIC provides for the output of numbers and strings. These facilities are provided by the various PRINT commands, namely PRINT, PRINT # and LPRINT.

All of these commands will print numbers and strings in Free Format, in which BASIC will:

- print numbers using the minimum number of characters possible.
- print strings literally, except for Tab expansion on the console and printer (but not on file output).
- organise output into 'print zones' if requested.
- generate a new line if a print item will not fit on the current line.

With the USING option these commands will print numbers and strings in formats dictated by a template string provided by the user, thus:

- numbers may be printed with a specified number of digits.
- numbers may be printed with commas after thousands, millions etc.
- numbers may be printed in a specified 'scientific' form.
- part or all of strings may be printed, with Tab expansion on the console and printer (but not on file output).
- print zones are ignored.
- new lines are generated only when a line is completely filled.

Printing to the console (PRINT), printing to the printer (LPRINT) and printing to a file (PRINT #) all work in the same way. To support 'print zones', tab expansion and the TAB function BASIC maintains a logical position by keeping track of the characters which have been output – see Section 6.3 below.

The console and printer have widths associated with them. BASIC will generate a new line if the logical position reaches the device width. In Free Format printing BASIC will generate a new line before printing an object if that object would not fit between the logical position and the device width. Files are treated as being infinitely wide. The console and printer widths are settable, and may be set infinitely wide.

6.1 Operation of a Print Statement

A print command may be followed by a number of print items, separated by either commas or semicolons. An item may be a string or numeric expression, or a TAB or SPC print function. Each expression is evaluated and the result printed in 'free format'. A comma following the expression causes BASIC to advance to the beginning of the next 'print zone' after the item is printed. A semicolon serves only to separate the items, and has no effect on the output.

Two special functions, TAB and SPC, are supported only in print commands. TAB prints spaces to move to the given print position. SPC prints a given number of spaces. TAB and SPC print functions are treated as if they were followed by a semicolon.

Formatted printing is introduced by the USING keyword, followed by a string expression which gives the format template. The template is followed by a number of expressions, whose values are to be printed. Commas or semicolons may be used to separate the expressions – neither has any effect on the output.

When all the arguments of a print command have been processed a new line is started, unless the command ends in a comma, semicolon or TAB or SPC.

6.2 Control Character Handling

When given a Tab (&H09) character to print on the console or printer BASIC usually transforms it into a number of spaces, sufficient to move forward to the next tab position – where tabs are implicitly set every eighth column. This action is independent of any characters which precede the Tab, so if the value &H09 forms part of an escape sequence BASIC is unaware of the fact, and transforms the Tab. The commands OPTION NOT TAB and OPTION TAB may be used to inhibit and re-enable, respectively, this tab expansion.

All other control characters are output as directly as possible, avoiding as much interference from the operating system as it will allow.

6.3 Logical Position

The logical position starts at column 1. Each time a character is printed the position is incremented by one, except for the following characters:

- Backspace (&H08), which counts as minus one.
- Tab (&H09) which if expanded into spaces counts 1 for each space.

- Carriage Return (&H0D), which resets the position to 1.
- Other control characters in the range &H00..&H1F, which count as 0.

The logical position may not correspond to the actual position after control sequences have been output.

6.4 Free Format Printing

The results of numeric expressions are printed as described below. The result of string expressions are output character by character after BASIC has checked that the string fits. The effects of SPC and TAB are described below.

A comma following a print item causes BASIC to space forward to the start of the next print zone before processing the next expression. A semicolon simply separates items.

6.4.1 Print Zones and Device Width

BASIC divides the output line into zones, whose width is set by the ZONE command, and which by default are 15 characters wide. When a print command spaces forward to the next print zone BASIC will start a new line if there are less than 'zone width' characters to the edge of the device.

Before printing any item BASIC checks that there is enough room for it between the current position and the edge of the device (as defined by the current WIDTH), unless the current position is the beginning of the line. If there is not enough room, a new line is started before the result is printed. When printing the result of a string expression BASIC uses the length of the resulting string when testing whether there is enough room. Note that this length is unlikely to correspond to the amount of room required for the string if it contains control characters.

6.4.2 Free Format Number Printing

Positive numbers are preceded by a space; negative numbers by a minus sign. All numbers are followed by a space.

All numbers are printed with the minimum of characters. No decimal point is printed if there are no significant fraction digits. At least one digit is printed before any decimal point – so numbers may include a leading zero.

Single length numbers are printed in scaled format (with an exponent) if they cannot be accurately represented by a number with seven or less digits (excluding a possible leading zero).

Double length numbers are printed in scaled format (with an exponent) if they cannot be accurately represented by a number with sixteen or less digits (excluding a possible leading zero).

6.4.3 SPC Print Function

SPC prints a given number of spaces, starting unconditionally at the current position. SPACES\$ is not quite equivalent, since BASIC would check that the string fitted on the current line, and possibly issue a carriage return before the spaces.

SPC need not be followed by a comma or semicolon, a semicolon is assumed (including when SPC is the last item in the print command).

6.4.4 TAB Print Function

TAB prints spaces to move to a given print position. If the required print position is at or after the current position, spaces are printed until the required position is reached (this may print nothing at all). If the required print position is before the current position, BASIC issues a new line followed by spaces to reach the required position on the new line.

TAB need not be followed by a comma or semicolon; a semicolon is assumed (including when TAB is the last item in the print command).

6.5 USING Formatted Printing

USING formatted printing is introduced by the USING keyword followed by a string expression giving the required format template. A number of numeric and string expressions may follow, separated by commas or semicolons. The expressions are evaluated in turn and then the template is processed to find a suitable format specification for each result. If the template string is exhausted while searching for a format specification the process is restarted from the beginning of the string. If no suitable format is found an error (Error 5) is generated.

Note that USING Formatted printing, unlike Free Format printing, does not perform any print zoning or checking for room on the current line.

6.5.1 Format Template

The format template is a string which is interpreted character by character to control the way in which the result of each expression is printed. The meaning of characters in the template is described below. The following characters are recognised in format specifications (Note: Character set dependent – see Appendix VIII) :

! \ & # . + - * \$ ^ ,

If other characters are encountered when searching the template for a format specification they will be printed and the search will continue. If one of the above characters is found out of context it too will be printed. The "_" character is not printed, but causes the following character to be printed "as is".

6.5.2 Format Specifications suitable for Strings

! Only the first character of the string is printed.

\spaces\ The first *n* characters of the string are printed, where the string \spaces\ is *n* characters long.

& The complete string is printed "as is".

6.5.3 Format Specifications suitable for Numbers

Numbers may be printed without an exponent part (unscaled) or with one (scaled). A format specification for a number may not exceed 24 characters, not counting the exponent and trailing sign options.

Numbers are rounded to the number of digits printed.

Note: The characters used here are character set dependent – see Appendix VIII.

The Body of a Number

Each # specifies a digit position. At least one # must be specified.

. Specifies the position of the decimal point. There may be at most one . in a number format specification.

, May appear before .; specifies a digit position, and also requests that digits before the decimal point be divided into groups of three, separated by commas.

Chapter 6: Introduction to Printing

Leading Dollar and Asterisk Options The following are mutually exclusive options. These options must be specified immediately before the body of the number:

- \$\$** Specifies two digit positions. Specifies that a \$ sign be printed immediately before the first digit or decimal point (after any leading sign). Note that the \$ will occupy one of the 'digit' positions.
- **** Specifies two digit positions. Specifies that any leading spaces be replaced by asterisks.
- **\$** Specifies three digit positions. Acts as the previous two options combined.

Sign Options The default is for – to be printed immediately before the number (and any leading dollar) if the number is negative, and no sign at all if the number is positive. The – will occupy one of the digit positions before the decimal point.

It is possible to specify that + be printed for positive numbers, or that the sign is to follow the number.

- +** Specifies that + or – is to be printed, as appropriate.
If the + appears at the beginning of the format specification, the sign is printed immediately before the number (and any leading dollar).
If the + appears at the end of the format specification, the sign is printed after the number (and any exponent part).
- May only appear at the end of a format specification. Causes a – to be printed if the number is negative, and a space to be printed if it is positive. This sign indication is printed after the number (and any exponent part).

Exponent Option

- ^^^** following the body of the number, and preceding any trailing sign indication, enables the exponent option. The four characters reserve space for the four character exponent part.

The body of the number is printed with the maximum possible number of digits before the decimal point, reserving one digit position for the sign (even if the number is positive) if no other provision is made for it.

Field Overflow If a number cannot be printed within the format given, BASIC attempts to get as close as it can to the required format, but precedes the result by a % to indicate field overflow.

No number whose absolute value is greater than or equal to 10^{24} can be printed in unscaled format. Any attempt to do so causes the number to be printed in free format preceded by % to indicate a format failure.

Control of output devices

This chapter describes the use of BASIC to access screen and printer facilities.

Screen and printer facilities are selected by special control sequences, which are either single-character control codes or sequences of two or more characters, starting with the ESC character. Screen codes can be used to:

- position the cursor
- erase individual characters, sections of a line, or sections of the screen
- insert lines between existing lines of text
- enable/disable the cursor blob

Printer codes can be used to:

- initialise the printer
- select the printer character set
- enter special print modes
- set page size, margin positions and tab positions
- select character pitch, character style and line pitch
- change print position

To access a particular device feature, the appropriate control sequence is sent to the device, either as an individual item or, more usually, within other output. The facility to send these characters is provided in BASIC by the Print commands, PRINT and LPRINT. PRINT is used to send control sequences to the screen; LPRINT is used to send control sequences to the printer.

The elements of the control sequences are handled by BASIC as single characters; they are interpreted at their destination. However, certain character handling facilities within BASIC need to be disabled to ensure that the codes are interpreted correctly – see Section 7.3 below.

7.1 Specification of control sequences

The codes used to control the screen and the printer are of two types:

- Single-character control codes with ASCII character values in the range &H00...&H1F
- Escape sequences made up of two or more characters, the first of which is the ESC character (&H1B)

The codes used to control the screen are defined by the operating system under which BASIC is being run. The codes used to control the printer are defined by the printer firmware.

Details of the control sequences should be given in the appropriate reference documentation (ie. computer user guide, operating system manual, printer reference manual). The characters of these codes will be presented in one of three forms:

- as the character itself, where this is a printable character
- by giving the character's special name
- by giving the character's ASCII value (normally as a decimal number)

7.2 Transmission of control sequences

Control sequences are sent to the appropriate output device by using Print commands. PRINT is used to send codes to the screen; LPRINT is used to send codes to the printer.

The characters of the codes should be sent as single characters, either as individual print items in the Print statement (separated by semicolons) or concatenated in character strings.

7.3 Interpretation of control sequences

Interpretation of the codes is handled at the destination device. Correctly specified codes will have the required effect; an incorrectly specified code can access a different device feature and/or insert spurious characters in the output.

The codes are output to the destination device as directly as possible. However, the following should be noted:

(i) BASIC normally replaces any &H09 (Tab) character by spaces sufficient to move forward to the next tab position. Where the &H09 character forms part of an escape sequence, BASIC is unaware of this fact and will transform the Tab character. The commands `OPTION NOT TAB` and `OPTION TAB` may be used respectively to inhibit and re-enable this tab expansion.

(ii) BASIC keeps track of the current print position by incrementing a logical print position; details of how this is done are given in Section 6.3. This logical position is used in conjunction with the device width to determine where BASIC should generate a new line.

Where control sequences are output, the logical position may not correspond to the actual position. Moreover, BASIC may generate the new line in the middle of a control sequence, causing the code to be incorrectly specified.

The new line is not generated if the device is seen as infinitely wide. This is achieved by setting the device width to the special value 255. Thus it is recommended that the command `WIDTH 255` is used before any series of screen control codes, and that the command `WIDTH LPRINT 255` is used before any series of printer control codes. A further `WIDTH/WIDTH LPRINT` command should be used to restore the device width.

Introduction to File Handling

A major part of BASIC is concerned with file handling. The facilities may be categorised:

- **Disc and Directory Operations**

In which BASIC operates on complete files or the names of files.

- **Sequential File Handling**

In which files are treated as a sequence of characters to be read or written only in that sequence. The facilities provided are variants of those for console input and output.

Sequential file handling will read and write simple ASCII text files.

- **Random File Handling**

In which files are treated as a collection of fixed size records, referred to by record number and which may be read or written in any order. A record may be organised as a number of named fields, or may be ASCII text.

Unless special steps are taken, random files produced by BASIC programs may not be suitable for processing by other programs.

- **Keyed File Handling (Jetsam)**

BASIC supports keyed file handling. Keyed files are similar to random files, in that the data is organised as fixed size records, and is read and written in the same way as random files. Access to the data is performed via an index in which user supplied keys are kept in key value order.

Keyed files may only be processed by Mallard BASIC programs.

- **Multuser File Handling (16-bit systems only)**

Where the operating system will support it, versions of BASIC may further include facilities for file and record locking for use in multiuser systems. Where they are provided these facilities supplement all supported file handling methods, not just keyed file handling.

This chapter describes all file handling other than keyed file handling, which is covered in Chapter 9. The keyed file facilities are an extension to the standard facilities, particularly random file handling, so this chapter is also relevant to keyed file handling.

8.1 File Names

Whenever BASIC is presented with a file name all characters are converted to their upper case equivalents and have Bit 7 cleared to zero (cf. `UPPER$` and `STRIP$` functions). File names may contain only ASCII printing characters in the range &H21...&H7E except for [] < > \ | = , ; ,

The "." character may be used to delimit file names and file types in the usual way. The ":" character may be used to delimit drive names and, on CP/M type operating systems, user numbers. Any leading and trailing spaces are accepted and removed, as are spaces around ":" and "." characters. The 'wild card' characters "?" and "*" are accepted by some commands, and have their usual meaning.

The MS-DOS 2 version of BASIC further supports MS-DOS 2 Tree-structured directories and path names. The "\" character may be used to delimit directory names in the usual way; alternatively, "/" may be used. These characters may be surrounded by spaces, which are ignored. Directory names may not include wild cards.

8.2 Disc and Directory Operations

These operations act on complete files or their names. Some of the commands have forms which are intended for use in Direct Mode, and which mimic the syntax of the equivalent operating system commands.

The `RESET` command has two effects. Firstly, it closes all open files. Secondly, it issues a general disc system reset to the operating system. If you are using real disc drives, you may need to issue a `RESET` before changing discs, particularly if data is to be written to a new disc.

The `DIR` and `FILES` commands produce a directory listing to the console. A file specifier parameter may be given, which may contain 'wild card' characters in the usual way, to produce a listing of part of the directory. `DIR` and `FILES` commands may be abandoned by typing Control-C. Typing Control-S will suspend the output, which may be restarted by a second Control-S. (Except in Run-Only versions where neither key has any effect).

The `FIND$` function returns the full name of the first or *n*th file whose name matches the given name. If no such file exists then `FIND$` returns an empty string. `FIND$` when given a name with 'wild card' characters may be used to read all or part of a directory, for a program to process in some way. `FIND$` when given an

unambiguous name may be used to check the existence of a file of that name. The name returned is the full name provided by the operating system, and may include Bit 7 markers on some characters.

The DEL, ERA and KILL commands may be used to delete files. The file name parameter may include 'wild card' characters, to delete a number of files, but for safety's sake the form *.* is rejected. The DEL command is only present in versions of BASIC for MS-DOS (PC-DOS) type operating systems. The ERA command is present only in versions of BASIC for CP/M type operating systems.

The REN and NAME commands may be used to change the name of a file. The commands take two file name parameters, the old name and the new name for the file. Neither parameter may contain any 'wild card' characters. If a drive name is given then both names must specify the same drive. A file with the old name must exist and a file with the new name must not.

The TYPE and DISPLAY commands read the given file and display it on the console. In Direct Mode these may be used to inspect the contents of a file. When a program is running these may be used to print a quantity of information (for example a help file) without it taking up space in the program. When reading the file a SUB (Control-Z, &H1A) character is taken as end of file. TYPE and DISPLAY commands may be abandoned by typing Control-C. Typing Control-S will suspend the output, which may be restarted by typing a second Control-S. (Except in Run-Only versions where neither key has any effect).

8.2.1 Tree-structured directories (MS-DOS 2 version)

The MS-DOS 2 version of Mallard BASIC includes extra commands and functions, specifically for handling MS-DOS 2's tree-structured directories.

Six commands directly mimic the equivalent operating system commands. MD and MKDIR commands are available for creating new directories; RD and RMDIR commands are available for deleting directories; and CD and CHDIR commands are available for changing the current directory.

Two functions are also included. The CHDIR\$ function returns a string containing the name of the current directory either on the default drive or on a specified drive. The FINDDIR\$ function looks for a given directory and, if it finds it, returns its name. If no matching directory is found, then a null string is returned.

8.3 Opening and Closing Files and File Numbers

Before processing a file it must be opened. The OPEN command performs this operation, in which the following must be specified:

i. **processing mode**

The file may be processed in one of the following ways:

- Sequential Input
- Sequential Output
- Random Access (Input and/or Output)
- Keyed Access (Input and/or Output)

ii. **the name of the file** (or files, in the case of keyed files)

This must be a valid file name, without 'wild cards'. Whether the file must or must not exist and other such restrictions depend on the processing mode.

iii. **a file number**

Once a file has been opened it is no longer referred to by name. The OPEN command associates the opened file with a file number, and all other commands use that number. The number to be used is specified in the OPEN command, and must be a number which is not in use at the time.

File numbers are small integers in the range 1...maximum, where the maximum is by default 3, but may be set to some other value when BASIC is first started, or at any time thereafter (using a MEMORY or CLEAR command).

iv. **a file lock**

A file lock may be specified (for keyed files it must be specified). In multiuser systems BASIC will attempt to gain the lock when opening the file. If the lock cannot be obtained the OPEN fails. If no lock is specified BASIC will attempt a default lock. The restrictions on the locks that may be applied and the default lock depend on the processing mode.

Other information may be required or may be specified for random and keyed file handling. See the relevant sections and the description of the OPEN command for more detail.

Once a file has been opened it can be read or written (or both) as allowed in the given processing mode. The sections which follow further describe file processing, except for keyed files and multiuser facilities, which are covered in Chapter 9.

When all required operations on a file have been performed it must be closed. Closing a file has three main effects:

- any output data buffered in BASIC is written away.
- the operating system is informed that all processing of the file is complete, so that any outstanding disc operations can be performed.
- the file number is released, and may later be re-used.

The CLOSE command may be used to close some or all open files. Other commands implicitly close all files (for example RESET and RUN). Closing keyed files have further effects which are discussed in Chapter 9.

8.4 Sequential File Handling

BASIC expects sequential files to be simple ASCII text files. The file may contain characters of any value in the range &H00...&HFF. On input only the following have any significance to BASIC:

- Nulls (&H00) are ignored, and are not passed to the program.
- Carriage Return (&H0D) is treated as end of line (except following a Line Feed – see below), and is not passed to the program, though end of line does have other effects. A Line Feed (&H0A) immediately following a Carriage Return is discarded.
- Line Feed (&H0A) is treated as a normal character (except following a Carriage Return – see above), and is passed to the program. A Carriage Return (&H0D) immediately following a Line Feed is discarded.
- SUB (&H1A, Control-Z) is treated as end of line and end of file. The EOF function may be used to test for end of file. It is an error to attempt to continue reading from the file once the end has been met.

On output BASIC puts the characters it is given directly to the file. The only characters added by BASIC are :

- Carriage Return, Line Feed (&H0D, &H0A) pairs for new lines.

When BASIC generates a new line this pair of characters is appended to the file. On output to a file BASIC only generates new lines when required at the end of output commands (such as PRINT #). This differs from output to the console or printer, where extra new lines may be inserted when the device width limit is reached or exceeded.

- SUB (&H1A, Control-Z) at end of file.

When the file is closed BASIC appends this standard end of file marker.

Note that, unlike console and printer output, BASIC never expands Tab (&H09) characters, but always sends them directly to the file.

Sequential input allows a file to be read starting with the first byte and proceeding to end of file and then stopping. It is not possible to change the order in which the bytes are read. The EOF function may be used to test whether end of file has been reached. All console input commands and functions have equivalents for reading files (except INKEY\$). INPUT # will read files as sequences of numbers in text form or strings of characters. LINE INPUT # will read a file line by line.

Opening a sequential output file creates a new, empty file – erasing any existing file of the same name. Characters output to the file are appended to it. When the file is closed a SUB (Control-Z, &H1A), end of file, character is appended. All console output commands have equivalents for writing to files. The WRITE # command is useful for the output of data which is to be read using INPUT #, particularly where strings are involved.

8.5 Random File Handling

BASIC random files are a collection of fixed size records, each with a unique number in the range 1...32767. Different files may have different record sizes, but the records within a single file are all the same size. Records may be read and written in any order. The contents of a record may be text, read and written using PRINT #, INPUT # and so on. Alternatively, the contents may be given a field structure, in which the bytes in the record are treated as fixed length strings.

When a random file is opened the record size must be specified (if it is not the default of 128 bytes is used). If no file of the given name exists an empty file is created. A file lock may be specified. On multiuser systems BASIC will attempt to acquire the given lock when opening the file, or will open the file unlocked if no lock is given – see Chapter 9. Once the file is open records may be read and written (unless the file is read locked, when records may only be read).

BASIC maintains one record buffer for each open random file. The commands GET and PUT transfer data to and from that buffer. The program may read from and write to the buffer in a variety of ways. Although BASIC does remember the number of the last record read or written, it does not enforce any correspondence between the contents of the buffer and any particular record. This means that the

contents of one record may be read and then written, changed or unchanged, to any other record, and that the contents of the buffer may be prepared and then written to a record without reading the record first.

Reading a record which has never been written is not recommended – the result is system dependent and may further be affected by the history of reads and writes to date. The first time a record is written space will be allocated for it, if required. Random record files are never reduced in size.

BASIC does not impose any limits on the use of record numbers. The position of the start of a record within a file is given by its record number minus one, multiplied by the record size. On some operating systems the existence of record n requires that all $n-1$ lower numbered records also exist, and occupy space on disc. When allocating numbers to new records, therefore, it may well be advantageous to use the lowest numbers available.

8.5.1 INPUT #, PRINT # and Related Commands

INPUT #, PRINT # and related commands may be used to read from and write to the random record buffer. A random record may, indeed, be treated as a number of lines by these commands.

BASIC maintains a pointer to the next byte to be read from the random record buffer. This pointer is set to the start of the buffer when the file is opened and subsequently after every GET and PUT command. Each time a byte is read or written the pointer is advanced. It is an error to attempt to read or write beyond the end of the record. When handled in this way each record of a random file may be regarded as a small sequential file, except that there is no equivalent of the end of file handling.

8.5.2 FIELD Commands and Related Commands and Functions

A FIELD command defines a template for the data in the random record buffer for the given file. Any number of such templates may be defined for a given file, allowing different record organisations within the same file. FIELD commands take the form:

FIELD file-reference, list-of: field

The file reference parameter specifies which open random file this template is for. Each field parameter takes the form:

field-size AS string-variable

The field size specifies the length of the field in bytes. The string variable is set to refer to the field. The first field parameter sets up a field at the start of the record, the second parameter sets up a field immediately following the first, and so on.

Execution of a FIELD command causes the current values of the string variables to be lost and replaces them by the current contents of the fields (note that this may override the setting of strings in earlier FIELD commands). The string variables may be used in expressions in the usual way, the value of a variable being the current contents of the field. To change the contents of a field the associated string may only be assigned to using one of:

LSET, MID\$ or RSET

because these commands are prepared to cope with assigning to a fixed length string. (A simple assignment to a string which is associated with a random record field breaks that association and creates a new string without changing the contents of the buffer. This allows the string variable to be re-used as a normal string once its use as a field is no longer required.)

In order to store numeric data efficiently a mechanism is provided to convert the internal representation of numeric data types to strings, and back again, as follows

numeric type:	Integer	Single Length	Double Length
function to create string:	MKIS	MKSS	MKDS
string length:	2	4	8
function to convert back:	CVI	CVS	CVD

Note that the strings produced are, in fact, copies of the internal form of the numbers, so numbers written this way are read back exactly as they were. This differs from other number input and output, where conversion between binary and decimal representations occurs.

Keyed File Handling and Multiuser File Handling

Keyed file handling is provided by Jetsam, a general purpose keyed file manager embedded within Mallard BASIC. This chapter briefly describes the facilities provided by Jetsam. The Jetsam commands and functions are given in detail in Chapter 10.

Multiuser versions of BASIC, under a suitable operating system, provide file and record locking facilities for keyed files and for ordinary random and sequential files.

For each keyed file Jetsam in fact maintains two files. Throughout this description the pair of files will be referred to as the 'keyed file'. The component files are called the 'data file' and the 'index file', the functions of which are as follows:

- **The Data File, which contains the user's data**

This is a standard BASIC random access file, except that the first 128 bytes (or so) of the file and the first 2 bytes of each record are reserved by Jetsam. When reading and writing data the usual GET and PUT commands are used, but the record numbers are provided by Jetsam, not by the user's program. The user should take the two reserved bytes into account when defining the record length. In all other respects these bytes are inaccessible to the user.

- **The Index File contains the keys**

This is a specially formatted file maintained by Jetsam. Each key entry in the Index File has three components:

- i. **Key Value**

Each key value is a BASIC string, up to 31 characters long. The key entries are kept in ascending order of key value (where keys are compared as described in Section 2.11.3). Several entries may have the same key value. A collection of key entries with the same key value is known as a 'key set'.

- ii. **Record Number**

Each key entry refers to a record in the data file. To find a record in the data file a program must first use Jetsam to look in the index for a key entry with the appropriate key value. Jetsam will return the record number which may then be used to read or write the associated data record.

iii. Rank

There are eight ranks of keys. Each rank may be regarded as an separate index for the data file. Alternatively the rank may be regarded as the first character of the key value, so all keys of rank 0 have values less than all keys in rank 1, and so on.

Jetsam commands and functions include parameters for all the file and record locking that will be required on multiuser systems.

Single-user versions of BASIC simply check that these parameters are in the legal range, but otherwise ignore them.

In multiuser versions of BASIC, the file and record locking parameters in Jetsam commands and functions are acted upon. In addition, some of the ordinary BASIC commands and functions take extra, optional, file and record locking parameters which apply to ordinary random and sequential files.

9.1 Consistency of Data and Index Files

There are periods while a keyed file is being written to during which the data and index files are not consistent with each other. The first time either file is written to (since the OPEN) Jetsam sets a marker in both. These markers are kept in the reserved area of the data file and in the index file header, and indicate that the files are in an indeterminate intermediate state. When the keyed file is closed the markers are cleared. Should the system fail before the keyed file is closed the markers will remain set. It is not possible to open a keyed file in which the markers are set, which ensures that Jetsam will not operate on inconsistent files.

Jetsam has a cache buffer scheme for the index files. Changes to the index file are made in main memory, delaying writing to the disc for as long as possible. The data file handling does not include any such buffering. It is quite possible, therefore, to make a number of changes without the index file on disc being altered while the data file on disc has been.

Apart from the simple, physical, inconsistency problem there may at times be logical inconsistencies in the keyed file while a program is modifying it, particularly when there are interdependencies in the data. The marking scheme will also guard against using incompletely modified data, provided the program ensures that the data is logically consistent before closing the file.

If several separate keyed files are logically linked the programmer should implement a similar marking scheme to ensure that the files are consistent with each other, given that Jetsam ensures that the individual files are internally consistent.

The marking scheme uses a generation number embedded in both data and index files. The files are marked inconsistent by increasing the generation number in one of the files by one and in the other by two. The files are only consistent when the generation numbers are equal. Before a keyed file is changed it is recommended that a copy of the data and index files is taken, so that in the event of a system failure the data can be restored to its last consistent state. (Note that the data and index files are marked as inconsistent with the back-up copies.) Closing the keyed file ensures that all modified data and index records are written to disc, and then clears the inconsistency markers.

The CONSOLIDATE function has the same effect as closing the keyed file and re-opening it, except that outstanding file and record locks are retained. In an application where changes to the keyed file occur infrequently compared to reading of the data it may be advisable to CONSOLIDATE the keyed file after each update (or set of updates) and immediately make a copy of the new files.

9.2 Multiuser Systems – Record and File Locking

When several users in a multiuser system simultaneously access the same file some control on their access is required to ensure that they are all presented with consistent data. This is straightforward if all users are only reading the file, but more complicated if the file is being changed.

When modifying a file in a multiuser environment the areas of change must be 'locked' until the modification has been completed, when the areas may be released. The locking prevents all other users accessing areas which are changing, where they would otherwise be in danger of accessing incomplete or inconsistent information.

When reading a file in a multiuser environment it may be necessary to ensure that the information does not change until some processing has been completed. In this case the areas which must not change may be locked to prevent any modification.

There are two levels of locking: File and Record. File locking is done when the file is opened, files may be opened unlocked, read locked or write locked. Record locking is required when a file is open unlocked to control access to records. Record locking is not required when a file is open read or write locked. Record locks cannot be applied in sequential file handling. Records may be read locked, write locked or unlocked.

Single user versions of Jetsam require lock parameters to be given legal values, but otherwise ignore them – in particular there is no check that the locks are sensible. Multiuser software is, therefore, accepted by single user versions. Software tested on a single user version should work on a multiuser system, but code for dealing with failed locks will not have been exercised and the lock settings will not have been rigorously checked.

9.2.1 Keyed File File Locks

Keyed files may be opened unlocked, read locked or write locked, as follows:

- | | |
|--------------|--|
| Unlocked | Other users may also open the file, but unlocked only. All users may read from or write to the file. Record locking may be required to control access to the file if it is being modified. |
| Read Locked | Other users may open the file, but read locked only. No user may write to the file, so no record locking is required. |
| Write Locked | No other users may open the file, and the open will fail if the file is already open. The owner of the write lock has exclusive access to the file and need not use any record locking.

Opening a file write locked does not commit the program to writing to the file. |

9.2.2 Random and Sequential File File Locks

Ordinary Random and Sequential files may be opened unlocked, read locked or write locked, as follows:

- | | |
|--------------|--|
| Unlocked | Other users may also open the file, but unlocked only. All users may read from or write to the file. Record locking may be required to control access to the file if it is being modified. |
| Read Locked | Other users may open the file, but read locked only. No user may write to the file, so no record locking is required. |
| Write Locked | No other users may open the file, and the open will fail if the file is already open. The owner of the write lock has exclusive access to the file and need not use any record locking.

Opening a file write locked does not commit the program to writing to the file. |

9.2.3 Record Locking in Keyed and Random Files

Records in keyed and random files may be unlocked, read locked or write locked, as follows:

Unlocked	There are no restrictions on what other users may do to the record or any of its keys. Any information about the record or any of its keys may be rendered invalid at any time.
Read Locked	<p>Other users may also read lock the record. The read lock will fail if the record is already write locked. No user may write lock the record.</p> <p>A record may not be altered nor may any keys be added or deleted while it is read locked.</p>
Write Locked	<p>No other users may lock the record. The write lock fails if any other user has locked the record. The owner of the write lock has exclusive access to the record.</p> <p>A record may not be altered nor may any keys be added or deleted unless it is write locked (or the file is write locked).</p> <p>Write locking a record does not commit the program to writing to it.</p>

With Jetsam keyed files record locks are applied when a key for the record is found, not when the record is read. This ensures that the record cannot be deleted in the time between extracting the record number from the index and accessing the record, nor can any other change take place to invalidate the operation.

With ordinary random files records may be locked and unlocked at any time using the LOCK function, and just before they are read and just after they are written by GET and PUT commands. GET and PUT when used with a random file will implicitly apply and release a temporary record write lock if the program does not explicitly lock records, see below.

9.2.4 Random File Temporary Record Write Lock

The temporary record write lock facilitates the processing of random files which are open unlocked by removing the need to explicitly lock records; provided that records are processed one at a time. BASIC will maintain a temporary write lock on one record of each open unlocked random file, as follows:

- **when a GET command is executed:**
 - i. any temporary write lock is dropped.
 - ii. any lock parameter is processed.
 - iii. if the record is not read or write locked then BASIC implicitly attempts to apply the temporary write lock.
 - iv. the record is read.
- **when a PUT command is executed:**
 - i. the record is written.
 - ii. if a lock parameter is present the required lock is applied and the temporary write lock forgotten.
 - iii. if no lock parameter is present any temporary write lock is dropped.
- **when a LOCK command is executed:**

if a temporary write lock exists for the record being locked, then it is forgotten when the new lock has been gained.

9.3 Finding Records and the Current Position in a Keyed File

The index file may be regarded as a list of key entries in ascending order of key value, with all keys for rank 0 before those for rank 1, and so on. While processing a keyed file Jetsam maintains a 'current position' which is the current key entry, if any. It is important to bear in mind that the current position is defined in terms of a key entry in the index, and although this position refers to a data record it does so indirectly.

When a keyed file is open unlocked on a multiuser system other users may change the index at any time. The record locking is provided so that areas of the keyed file may be held constant while they are processed, even though the rest of the file may be changing. While a position is defined in terms of a key entry it is the record which is locked, which means that the record and all keys associated with it are locked simultaneously.

To access data in the keyed file the program must use Jetsam commands to move the current position within the index, and may then GET and PUT the records whose numbers are extracted from the key entries. The program may remember positions in the keyed file for future use (provided, on multiuser systems, that suitable locks are maintained).

Various facilities are provided to move the current position within the index:

- move to the beginning of the given rank.
- reset the current position to a previously stored value.
- find the first key entry with the given value (in the given rank).
- move to the next key entry.
- move to the previous key entry.
- move to the next key set.

These functions support both random and sequential processing of the keyed file, in which data is accessed and ordered by key value.

On a multiuser system if the current position is not locked the key entry to which it refers may be deleted at any time by another user. Each time a seek from current position function is executed Jetsam first checks that the key entry still exists.

Under some conditions the current position is not set, or is unset. That is to say it does not correspond to any key entry.

9.4 Overview of Facilities and Recommended Use

This section is a brief overview of the facilities provided by Jetsam, with some recommendations on their use. Some ordinary BASIC commands and functions are extended for keyed file handling and multiuser file handling. All commands and functions are described in detail in Chapter 10.

9.4.1 Jetsam Sizes and Restrictions

The number of keyed files open at the same time is restricted only by current maximum file number in the usual way (see the MEMORY and CLEAR commands and Appendix I).

The minimum keyed data file record length is 2, both of which are reserved for use by Jetsam. The maximum record length is set in the usual way (see the MEMORY and CLEAR commands and Appendix I). The record length should be set to 2 more than the sum of the item lengths in the FIELD command.

The maximum data file record number is 32767, as it is with all BASIC random files. The first 128 bytes of the data file is reserved for use by Jetsam, so the first record number depends on the record length:

```
IF record length >= 128 THEN first record number = 2
ELSE first record number = 2 + INT(127/record length)
```

The maximum size of keyed file that Jetsam can manage is restricted by the maximum record number and the record size, but thereafter only by the operating system and available resources.

Key values are strings of up to 31 bytes. Keys are compared on a byte by byte basis from left to right. UPPER\$ or LOWER\$ may be used to ensure that the case of characters in a key is consistent. Strings produced by MKI\$, MKS\$ and MKD\$ are not really suitable for use as keys (but see Chapter 10).

Eight ranks are supported, numbered 0 to 7.

Jetsam maintains a cache for index files. The size of this cache may be changed using the BUFFERS command.

In multiuser systems BASIC requires some space to keep information about each active record lock. The maximum number of record locks per file is set by the second parameter of the BUFFERS command.

9.4.2 Creation of a Keyed File

CREATE, RANKSPEC

CREATE Before a keyed file may be written to, the data and index files must be created and initialised. The CREATE command creates both files, initialises the data file's header and sets up an empty index. CREATE immediately opens the newly created file. When a new file is OPENed for output or for random access BASIC automatically creates it. This is not true of keyed files which must exist before they may be OPENed.

RANKSPEC Each rank in a keyed file may be marked so that a new key of the same value as an existing key may not be added. RANKSPEC sets the 'duplicates allowed' status of a given rank.

9.4.3 Opening and Closing Keyed Files and Files in Multiuser Systems

OPEN, CLOSE, CONSOLIDATE, BUFFERS

OPEN The OPEN command includes a keyed mode of access. The keyed OPEN requires both the data and index file names, as well as the lock requirement. The operation will fail if either file does not exist, if the files are inconsistent or if the lock is not granted.

CLOSE It is most important to CLOSE a keyed file when the program has finished with it. If the file has been written to, all buffered data and index records are written to disc, and the file is marked consistent. All record locks for the file and any file lock are released. If the file is not explicitly closed the files will not be marked consistent and any locks will persist.

For use in multiuser systems the OPEN command when applied to ordinary random and sequential files takes an optional file lock parameter. The CLOSE command when applied to random and sequential files will release any record locks.

CONSOLIDATE The CONSOLIDATE function may be applied to an open keyed file causing all buffered data and index records to be written to disc, and the file to be marked consistent, without closing the file or releasing any record locks.

BUFFERS Jetsam maintains a cache for index files. The size of this cache may be changed using the BUFFERS command. The larger the cache the more chance Jetsam has of keeping index records in memory, which improves performance. The store given over to the cache is subtracted from the store available to the program. The BUFFERS command allows a suitable balance to be chosen.

In multiuser systems BASIC requires some space to keep information about each active record lock. The maximum number of record locks per file is set by the second parameter of the BUFFERS command. (Note that the default maximum is zero, so a BUFFERS command must be issued before any record locking can be done.)

9.4.4 Reading, Writing and Locking Data Records

GET, PUT, LOCK

Keyed files The usual GET and PUT commands work with keyed files. The way in which these commands should be used is a little different to their use with ordinary random access files:

i. Creation of Records

In ordinary random access files a new record is created by preparing its contents and then writing it using a PUT command, in which the new record's number is implicitly or explicitly set. The procedure is similar for keyed files, except the new record must be written using an ADDREC function, in which Jetsam allocates the record number and simultaneously enters a key for the new record into the index.

ii. Record Number

Record numbers are under Jetsam's control, and serve only as pointers from the keys to their associated data. Only record numbers returned by Jetsam may be used by the program. If a program is to be used in a multiuser system it must only retain knowledge of those records for which it has read or write locks.

iii. Record Update

Because Jetsam uses the first two bytes of each data record the program must firstly avoid these bytes, and secondly it must GET the record before modifying it, even if none of the user's portion of the record will remain unchanged.

When a record is PUT, a check is made to ensure that the information in the first two bytes is still valid. The operation of ADDKEY and DELKEY alter this information, so it is important that no ADDKEY or DELKEY is performed between a GET and a PUT to the same record.

iv. Write Locks

In a multiuser system the program must not write to a record which it does not have write locked.

The LOCK function may be used to change the lock state of a record. The record must already be locked, so this function may only be used to change between read and write locks, or to unlock the record.

Random Files in Multiuser Systems For use in multiuser systems GET and PUT commands for ordinary random files take an optional record lock parameter. In the case of GET the lock is applied immediately before the record is read. In the case of PUT the lock is applied immediately after the record is written.

The LOCK function may also be used with ordinary random record files. If the program does not apply explicit record locks BASIC will implicitly apply a temporary write lock before GETting a record and release it after PUTting it – see Section 9.2.4, above.

9.4.5 Creating and Deleting Records and Keys: Keyed Files Only

ADDREC, ADDKEY, DELKEY

ADDREC creates a new record in the data file and inserts the given key for it into the index. Jetsam allocates a record number for the new record. Records may not be created in any other way. The creation of the new record and the addition of a key for it must be done together, otherwise there would be no means of accessing the record. The data for the new record is taken from the current random record buffer; Jetsam initialises the first two bytes which are reserved for its use.

ADDKEY A record may have several keys, in several ranks. **ADDREC** sets the first key for each new record, **ADDKEY** may be used to set additional keys.

DELKEY removes a given key for a given record from the index. If the only key for a record is deleted then the record is also deleted, and the area occupied by the record may be used by a subsequent **ADDREC**. To change a key value, therefore, **ADDKEY** should be used to insert the new key before **DELKEY** is used to remove the old key.

9.4.6 Finding Keys & Changing the Current Position: Keyed Files Only

SEEKRANK, SEEKKEY, SEEKNEXT, SEEKPREV, SEEKSET, SEEKREC

All these functions attempt to find a key entry, specified in a variety of ways. If successful, the key entry found becomes the current position and the given lock is applied to the associated data record. Note: these functions do not read or write data; to do this use GET and PUT.

SEEKRANK searches for first key of the given rank. **SEEKRANK** positions Jetsam ready for sequential processing of the data with keys in that rank.

SEEKKEY searches for first key of the given rank and the given value. **SEEKKEY** provides random access to the data records by key value. Note that the key found may be the first of a key set.

SEEKNEXT searches for the next key after the current or the given position. **SEEKNEXT** may be used to process forwards through the keyed file.

SEEKPREV searches for the key immediately before the current or the given position. **SEEKPREV** may be used to process backwards through the keyed file.

SEEKSET searches for the first key of the next key set after the current or the given position. **SEEKSET** is similar to **SEEKNEXT**, except that it skips past keys which have the same value (and rank) as the key from which the search starts.

SEEKREC searches for a particular key value in a particular rank which references a particular record. **SEEKREC** may be used to move back to a remembered position – noting that on multiuser systems the program should have the record locked.

9.4.7 Finding the Current Position: Keyed Files Only

FETCHREC, **FETCHKEY\$**, **FETCHRANK**

These functions each return one of the components of the current position.

FETCHREC returns the record number of the record referenced by the key at the current position in the index. While the record is read or write locked the record number may be used to access the data record.

FETCHKEY\$ returns the key value of the current position.

FETCHRANK returns the key rank of the current position.

9.5 General Remarks on Jetsam Commands and Functions

Most Jetsam operations are presented as functions rather than commands. This is because they return a code indicating the result of the operation. To perform such a Jetsam operation, therefore, the function must appear as part of an expression! A statement of the form:

```
return.code% = SEEKNEXT(#1, 1)
```

is recommended, though there are cases where the function may usefully appear in the conditional expression in an IF.

In the example given, Jetsam will attempt to seek the next key entry in file #1 and apply a read lock to it. The operation may succeed, but it may be important to the program that the key found is in the next key set, or in the next rank. The operation may fail for a variety of reasons. Each of the possible results produces a unique return code, which may be used by the program to decide what to do next.

9.5.1 Errors and Return Codes

Errors detected while executing a Jetsam command or function are signalled in the usual way, and ON ERROR GOTO may be used to trap them. Examples of errors are:

- incorrectly specified parameters
- illegal operations
- attempting to read or write records without the required locks
- general disc problems, such as disc full

Jetsam errors are given in Appendix II along with all other error messages.

The values of the return codes produced by the Jetsam functions and their general meanings are as follows:

0	unqualified success.
101	moved past end of key set.
102	moved past end of rank.
103	moved past last key in index.
105	key not found.
115	no current position.
130	file is read locked by another user.
131	failed to write lock the record of the key to be added or deleted.
132	could not read lock the record.
133	could not write lock the record.

The possible return codes for each function are given in the descriptions in Chapter 10. The precise meaning of a return code is sometimes dependent on the function – in particular, whether the return codes are designated Success or Failure codes. In general if a Failure code is returned by a function then no change will have been made to the files, the current position or the lock state. Occasionally a Failure does have side effects, in which case they are explicitly stated.

If a function returns a 'lock failure' (return codes 130...133) then the recommended actions are as follows:

- 130 The file is locked by another user. Report the fact to the operator and await further instructions, since the other user may leave it locked for some time.
- 131 When adding or deleting a key Jetsam has to write to the data record. If the record is not already write locked (and the file is not write locked) Jetsam will attempt to temporarily write lock it. This return code is produced if Jetsam cannot write lock the record. The record may be locked by another user, and the lock may soon be released. The program should repeat the operation a few times before seeking operator intervention.
- 132,133 The ADDKEY, DELKEY, LOCK and the various SEEK functions have a *lock* parameter. If this lock cannot be achieved the record may be locked by another user, and the lock may soon be released. The program should repeat the operation a few times before seeking operator intervention.

Chapter 10

The Commands and Built-In Functions

This chapter contains a detailed description of all Commands and Built-In Functions, and is arranged in alphabetical order of keyword.

10.1 Definition of Common Terms

The following terms are used throughout the section (Note: the special characters included in these definitions are character set dependent – see Appendix VIII) :

<i>address-expression</i>	A <i>numeric-expression</i> which is converted to Unsigned Integer – see Section 2.10.
<i>array-variable-name</i>	The name of an array, which may include its <i>type-marker</i> .
<i>expression</i>	An expression yielding a value of any type.
<i>file-name</i>	A collection of characters which will be recognised as a file name by the operating system. BASIC forces lower case characters to their upper case equivalents before presenting the name to the operating system.
<i>file-name-expression</i>	A <i>string-expression</i> whose value is a valid <i>file-name</i> .
<i>file-number-expression</i>	An <i>integer-expression</i> which yields a value in the range 1.. <i>files</i> , where <i>files</i> is the maximum file number set when BASIC was initialised (see Appendix I).
<i>file-reference</i>	Is: [#] <i>file-number-expression</i>
<i>integer-expression</i>	A <i>numeric-expression</i> , which when rounded to an integer yields a value in the range -32768 to +32767.

<i>line-number</i>	A decimal number in the range 0...65534. In Direct Mode a dot may be used instead, meaning the current line.
<i>line-number-range</i>	A <i>line-number-range</i> specifies all lines whose numbers are within the inclusive range given. The range may take one of the forms :
<i>line-number</i>	Only the given number is in the range.
<i>line-number–line-number</i>	Lines from the first number to the second inclusive.
<i>line-number–</i>	Lines from the given line to the end of the program.
<i>–line-number</i>	Lines from the beginning of the program to the given line.
<i>logical-expression</i>	BASIC does not support a separate boolean type. Integer value 0 is treated as false, and any other Integer value as True. A <i>logical-expression</i> is, therefore, an <i>integer-expression</i> by another name.
<i>numeric-constant</i>	See Section 2.4.
<i>numeric-expression</i>	<p>An expression which returns a value of any of the numeric types – Integer, Single Length and Double Length.</p> <p>Note that this includes expressions previously categorised as relational and logical.</p>
<i>numeric-variable</i>	A reference to a variable, which may be an array item, whose type is one of the numeric types.
<i>quoted-string</i>	A <i>quoted-string</i> is between 0 and 255 characters enclosed in double quotes, or starting with double quotes and terminated by carriage return.
<i>record-number</i>	An <i>integer-expression</i> yielding a value in the range 1...32767. Records in random and keyed (Jetsam) files are referenced by <i>record-number</i> .
<i>simple-variable</i>	A reference to a variable excluding array variables. The reference may include a <i>type-marker</i> .

<i>string-expression</i>	An expression which yields a value of type String.
<i>string-variable</i>	A reference to a variable, which may be an array item, whose type is String.
<i>type-marker</i>	One of the characters : % indicating Integer ! indicating Single Length # indicating Double Length \$ indicating String
<i>variable</i>	A reference to any form of variable. This includes variables which are items in an array, in which case suitable subscripts must be included.
<i>variable-name</i>	The name of a <i>simple-variable</i> , which may include its <i>type-marker</i> .

10.2 Definition of Common Terms Used Only in Jetsam and in Multiuser versions

The following further terms are used in the descriptions of Jetsam commands and functions, or standard commands and functions which are extended when Jetsam is being used.

<i>index-position</i>	is: <i>rank</i> , <i>key-value</i> , <i>record-number</i> The values of the component parts of an <i>index-position</i> must previously have been returned by Jetsam and, preferably, refer to a record which is locked.
<i>key-value</i>	A <i>string-expression</i> yielding a string whose length is in the range 0...31.
<i>lock</i>	An <i>integer-expression</i> specifying a record or file lock. The expression must yield a value in the range 0...2, which is interpreted thus: 0: no lock or unlocked 1: read lock 2: write lock
<i>rank</i>	An <i>integer-expression</i> yielding a value in the range 0...7, which specifies one of the eight possible ranks of keys.

<i>drive-spec</i>	is:	<i>letter</i> :
<i>path</i>	is:	<i>[\] dir-simple [\ dir-simple]*</i>
	or:	<i>\</i>
<i>file-name</i>	is:	<i>[drive-spec] [path \] file-simple</i>
<i>dir-name</i>	is:	<i>[drive-spec] path [\]</i>
<i>dir-simple</i>	is:	<i>.</i>
	or:	<i>..</i>
	or:	<i>name-simple</i>
<i>file-simple</i>	is:	<i>name-simple</i>
<i>name-simple</i>	is:	<i>name-part [type-part]</i>
<i>name-part</i>	is:	<i>file-char [file-char]*</i>
<i>type-part</i>	is:	<i>.</i> <i>[file-char]*</i>

ABS

Absolute value.

Function

Use.

To determine the absolute value of a given expression.

Form.

ABS (*numeric-expression*)

Notes.

ABS of a negative value returns that value negated. ABS of a positive value returns the value unchanged.

Associated Keywords.

SGN

ADDKEY

Add a new key for an existing record.

Jetsam Function

Use.

To add a new key to the index file corresponding to a record which already exists in the data file. The current key is set to this key.

Form.

ADDKEY (*file-reference*, *lock*, *rank*, *key*, *record-number*)

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies what lock to apply to the record once the new key has been added.

The *record-number* specifies which record to add a new key for. The number must previously have been returned by Jetsam, and the record must be write locked.

The *rank* and *key* parameters specify the rest of the new key entry to be inserted into the index.

Return Codes.

0	Success	the key has been added, the current position has been set to the new key entry and the required lock obtained.
116	Failure	a key of the given value already exists in the given rank, and that rank has been marked to reject duplicates (by RANKSPEC). The current position has been set to the first key of the given value in the rank, but no lock has been applied.
130	Failure	the file is read locked by another user.
131	Failure	could not write lock the record of the key to be added.
132	Failure	could not apply the read <i>lock</i> to the record.
133	Failure	could not apply the write <i>lock</i> to the record.

Notes.

If the record to be referred to by the new key entry is not write locked (and the file is not write locked) ADDKEY automatically obtains a temporary write lock. If this write lock cannot be obtained return code 131 is generated.

When the record was added to the file one key was inserted in the index for it. ADDKEY may be used to insert a second or subsequent key for the same record (for instance in another rank, where each rank is used as a separate index) or may be used to change a key by inserting a new one and using DELKEY to remove the previous one. (Note that the order ADDKEY then DELKEY is important – see DELKEY below.)

If successful the current position is set to the key just added and the required *lock* applied, otherwise the position is unchanged except when the failure is 116.

Associated Keywords.

ADDREC, DELKEY, RANKSPEC

ADDREC

Add a new record and key.

Jetsam Function

Use.

To add a new key to the index file and a corresponding new record to the data file. The current position is set to this key.

Form.

ADDREC (*file-reference*, *lock*, *rank*, *key-value*)

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies what lock to apply to the new record.

The *rank* and *key* parameters form part of the key entry inserted in the index. The record number is supplied by Jetsam.

Return Codes.

0	Success	the key and record have been added, the current position has been set to the new key entry and the required lock obtained.
116	Failure	a key of the given value already exists in the given rank, and that rank has been marked (by RANKSPEC) to reject duplicates. The current position has been set to the first key of the given value in the rank, but no lock has been applied.
130	Failure	the file is read locked by another user.
131	Failure	could not write lock the record being added.

Notes.

The current contents of the file's random record buffer are written to the new record. Jetsam supplies the record number.

If the operation is successful the current position is set to the new key entry, otherwise it is unaffected, except when the failure is 116.

This is the only way to add data to the keyed file. Unlike other forms of file, PUT may not be used, because Jetsam must supply the record number and must initialise the first two bytes of the record.

An error 131 may arise on operating systems where locking one record implicitly locks one or more other (usually adjacent) records. When Jetsam has allocated a record to this new data an attempt is made to write lock it, before writing the data. This attempt may fail.

Associated Keywords.

ADDKEY, DELKEY, PUT, RANKSPEC

Get ASCII value of character.

Function

Use.

To get the numeric value of the first character of a string, given that the ASCII encoding of characters is used.

The inverse of CHR\$.

Form.

ASC (*string-expression*)

where the *string-expression* yields a string at least one character long.

Notes.

The ASCII character set is listed in Appendix VIII.

Associated Keywords.

CHR\$

Arc-tangent.

Function

Use.

To calculate the arc-tangent of a given value. The result returned is in radians in the range $-\pi/2 \dots +\pi/2$

Form.

ATN (*numeric-expression*)

Notes.

While the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. ATN returns a Single Length result.

Associated Keywords.

SIN, COS, TAN

Automatic line numbering

Command

Unavailable in Run Only versions

Use.

To aid the input of program lines. Causes BASIC to generate line numbers.

Form.

AUTO [*line-number*][, [*increment*]]

If the *line-number* is omitted 10 is assumed.

If the *increment* and the comma are omitted then an increment of 10 is assumed. If the *increment* is omitted, but the comma is present, then the increment from the last AUTO command is used – unless this is the first AUTO command, when an increment of 10 is assumed. An *increment* of 0 is illegal (Error 5).

Notes.

While AUTO is running it generates line numbers for program entry. Each time AUTO generates the next line number the line editor is entered (see Chapter 4). If a line with that number already exists then it is presented for editing. If no line exists then a new empty line with the new line number is presented.

AUTO continues until the editing of the current line is abandoned rather than finished normally. Leaving AUTO returns BASIC to Direct Mode.

BUFFERS

Set the amount of memory for use by Jetsam.

Jetsam Command

Use.

To control the amount of memory reserved by Jetsam for its index cache, and, on multiuser systems, to control the amount of memory used by BASIC's lists of outstanding record locks.

Form.

`BUFFERS buffer-count[, lock-count]`
or `BUFFERS , lock-count`

The *buffer-count* is an *integer-expression* which gives the number of buffers that Jetsam may use, and must yield a result in the range 0...255.

The *lock-count* is an *integer-expression* which gives the maximum number of locks which will be set at the same time on each file, and must yield a value in the range 0..255. The *lock-count* is ignored on single user systems.

Notes.

In the first form of the command, if the *lock-count* parameter is omitted then the maximum number of record locks is unchanged. In the second form of the command the number of buffers used by Jetsam is not changed.

The more buffers allocated to the Jetsam index cache the better. At least six is recommended. Each Jetsam cache buffer requires 128 bytes of memory. The space used is subtracted from the space available to the program. The cache is shared by all keyed files open at any time. The memory used by the cache remains reserved even if no keyed files are open.

While the syntax of the BUFFERS command will accept *buffer-counts* up to and including 255, in practice the maximum number of buffers that may be set is limited by the store requirement.

When BASIC is first loaded no buffers are allocated to Jetsam. A program using Jetsam must, therefore, contain at least one BUFFERS command.

The number of cache buffers may be changed at any time, but requires less work if there are no keyed files open.

When BASIC is first loaded the maximum number of record locks is zero. To use any record locks, therefore, a suitable BUFFERS command must be issued. This applies to locking records in ordinary files, even though no index cache buffers are required. The operating system may impose an independent limit on the number of outstanding record locks.

Each record lock requires 7 bytes of memory per file. The space used is subtracted from the space available to the program. The memory remains reserved when no locks are in use.

Changing the maximum number of record locks causes all files to be closed, without marking any open keyed files as consistent. Keyed files should, therefore, be closed before this is done.

Associated Keywords.

CLEAR, CLOSE, OPEN, MEMORY

Call external subroutine.

Command

Use.

Allows externally developed subroutines to be invoked from BASIC.

Form.

`CALL simple-variable [(argument-list)]`

The *simple-variable*'s value gives the address of the subroutine. The *simple-variable* must have been set to the result of an *address-expression*.

The *argument-list* is a *list-of: variable*. The addresses of the variables in the argument list are passed to the subroutine.

Notes.

See Appendix III for a detailed discussion of the calling sequence, variable formats etc.

In Mallard-86 the address given is the offset part of the full address of the subroutine. The segment part is given by the User Segment Base most recently set by DEF SEG.

Associated Keywords.

DEF USR, USR, UNT, DEF SEG

Change the current directory.

Command

Use.

For Mallard-86 MSDOS 2 version, this command sets the current directory of the specified, or default drive.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all ignore this command.

Form.

CD *dir-name*

where *dir-name* specifies the directory to use as the current directory on the specified drive.

This command takes the rest of the current line as its argument, irrespective of colons or single quotes.

Notes.

The current directory is used implicitly in all *paths* that do not start from the root, ie. start with a\.

Associated Keywords.

CHDIR, MKDIR, RMDIR, MD, RD, CHDIR\$, FINDDIR\$

Convert to Double Length.

Function

Use.

To take a given expression and return its value as a Double Length number.

Form.

CDBL (*numeric-expression*)

Notes.

Many decimal fractions do not have exact binary floating point representations. Converting a single length floating point number to a double length one will highlight the effects of this. For instance:

```
100 DATA 0.1
110 READ A!
120 PRINT A!, CDBL(A!)
```

produces:

```
.1      .1000000014901161
```

Associated Keywords.

CSNG, CINT, CVD, MKD\$

CHAIN CHAIN MERGE

Chain into new program.

Command

Use.

Allows one program to load and enter another, optionally retaining some or all of the current program and variables.

Form.

```
CHAIN string-expression[ , line-number-expression]  
or CHAIN string-expression , [line-number-expression] , ALL  
  
CHAIN MERGE string-expression[ , [line-number-expression]  
or CHAIN MERGE string-expression , [line-number-expression] , ALL  
or CHAIN MERGE string-expression , [line-number-expression][ , ALL] ,  
DELETE line-number-range
```

Where:

The *string-expression* gives the name of the file containing the required program. If no type extension is given then '.BAS' is assumed.

The *line-number-expression* takes the form of an *address-expression*, which gives the line number to start executing from once the new program is loaded, as follows:

0 or absent	execution starts at the lowest numbered line.
1...65534	execution starts at the given line, if the line does not exist an Error 8 results.
65535	returns to System level.

ALL causes all variables in the current program to be retained. If ALL is omitted then no variables are retained, unless otherwise specified in COMMON statements in the current program.

DELETE causes the lines in the *line-number-range* given to be removed from the current program before the new program is loaded. (DELETE may be specified with the CHAIN command, but is wholly redundant.)

Notes.

RENUM will renumber the line numbers in the delete *line-number-range*, but will not affect the *line-number-expression*. In CHAIN this expression refers to a line in a separate program, so cannot be renumbered. In CHAIN MERGE BASIC cannot tell whether the *line-number-expression* refers to a line which will be unaffected by the merge, so it is left to be consistent with CHAIN.

The differences between CHAIN and CHAIN MERGE are as follows:

CHAIN

Deletes all current program lines.

Resets all DEFINT, DEFSNG, DEFDBL & DEFSTR settings. (So any common variables should have explicit types, or the new program must re-issue suitable DEF commands.)

Resets OPTION BASE setting, unless any arrays are passed to the new program.

CHAIN MERGE

Deletes only those lines specified in the DELETE clause (if any).

Retains all DEFINT, DEFSNG, DEFDBL & DEFSTR settings.

Retains OPTION BASE setting.

Actions common to CHAIN and CHAIN MERGE are:

- All User Functions are forgotten.
- ON ERROR GOTO is turned off.
- All open files are retained.
- RESTORE action is taken.
- All active FOR, WHILE and GOSUB commands are forgotten.

During a CHAIN MERGE operation a line in the new program with the same number as an existing line will replace the existing line.

If a protected program is CHAIN MERGED with an unprotected one, the result is a protected program.

Associated Keywords.

COMMON, LOAD, SAVE, RUN, MERGE, COMMON RESET

Change the current directory.

Command

Use.

For Mallard-86 MSDOS 2 version, this command sets the current directory of the specified, or default drive.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all ignore this command.

Form.

CHDIR *string-expression*

where the *string-expression* yields a *dir-name* which specifies the directory to use as the current directory on the specified drive.

Notes.

The current directory is used implicitly in all *paths* that do not start from the root, ie. do not start with a \.

Associated Keywords.

MKDIR, RMDIR, CD, MD, RD, CHDIR\$, FINDDIR\$

Return the current directory.

Function

Use.

For Mallard-86 MSDOS 2 version, this function returns a string containing the current directory of the specified drive.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all return the string "\".

Form.

CHDIR\$ [(*string-expression*)]

where the *string-expression* yields a *drive-spec* specifying the required drive.

If the (*string-expression*) is omitted then the default drive is assumed.

Notes.

The returned string is: \ [*path* \]

The returned string does not contain a *drive-spec*, always starts with \, and, if there is a *path*, terminates with a \.

Associated Keywords.

CHDIR, MKDIR, RMDIR, CD, MD, RD, FINDDIRS

Convert to character.

Function

Use.

To convert a numeric value to its character equivalent, given that the ASCII coding of characters is used.

The inverse of ASC.

Form.

CHR\$ (*integer-expression*)

where *integer-expression* must yield a value in the range 0...255.

Notes.

The ASCII character set is listed in Appendix VIII.

Associated Keywords.

ASC

Convert to integer.

Function

Use.

To convert the given value to integer representation, if possible.

Form.

CINT (*numeric-expression*)

The *numeric-expression* is rounded to integer, which must then be in the range -32768...32767.

Associated Keywords.

CSNG, CDBL, MKI\$, CVI, INT, FIX, ROUND, UNT

CLEAR

Clear all variables and files.

Command

Use.

To set all variables to zero if numeric, or null if string. Erase all arrays. Forget all User Functions. Close all files. This command may also be used to set the stack size and total memory available to BASIC as well as the maximum number of files and the maximum random record size.

Form.

CLEAR

or CLEAR, *high-memory*

or CLEAR, [*high-memory*], *stack-size*

or CLEAR, [*high-memory*], [*stack-size*], *number-of-files*

or CLEAR, [*high-memory*], [*stack-size*], [*number-of-files*], *maximum-record-length*

High-memory is an *address-expression* which gives the address of the highest byte in memory which may be used by BASIC. If omitted the current setting is retained.

Stack-size is an *address-expression*, which gives the number of bytes that BASIC should use for its stack. If omitted the current setting is retained. If present, the expression must yield a value of at least 256.

Number-of-files is an *address-expression* which gives the maximum number of files that may be handled at once. If omitted the current setting is retained.

Maximum-record-length is an *address-expression* which specifies the maximum random record size. If omitted the current setting is retained.

Notes.

It is only necessary to give values for those settings which are to be changed. If any values are given, a comma is required before the first parameter.

The optional parameters allow maximum number of files, the maximum random record size and total memory size to be set in much the same way as in the initial command line (see Appendix I).

CLEAR causes BASIC to forget any active FOR, WHILE or GOSUB commands.

BASIC's stack is used during expression evaluation and to hold information for FOR and WHILE loops and for GOSUB returns. The default size of 512 bytes should be adequate for most programs, unless extremely deeply nested FOR or WHILE loops or GOSUBs are used.

Associated Keywords.

MEMORY, HIMEM, FRE, OPEN, COMMON RESET

CLOSE

Close one, or more, files.

Command

Use.

To finish using files.

Form.

`CLOSE [list-of: file-reference]`

Each *file-reference* gives the number of a file to be closed.

If the file number list is omitted all files are CLOSEd.

Notes.

When a sequential output file is closed all outstanding data is written to the file.

After a CLOSE the file numbers involved no longer have any file associated with them, and they may be re-used.

CLOSEing a file number which is not in use has no effect, and is ignored.

Jetsam Extensions.

For Jetsam keyed files CLOSE closes both index and data files.

In multiuser systems CLOSE releases any remaining file or record locks for keyed, random or sequential files.

This operation is most important since it ensures that all data and index records are correctly written to disc and marks the data and index files consistent.

NB: Keyed files which are not explicitly closed using a CLOSE command will not be marked consistent. The following commands implicitly close all files, and will leave keyed files marked inconsistent:

BUFFERS (if the maximum number of record locks is changed)

CLOSE (without parameters)

SYSTEM

RUN (where a program is being restarted)

Associated Keywords.

OPEN, RESET, CONSOLIDATE (Jetsam), CREATE (Jetsam)

COMMON

**Declare variables to be retained during
CHAINing or COMMON RESET.**

Command

Use.

COMMON statements allow variables to be retained selectively past CHAIN, CHAIN MERGE and COMMON RESET operations.

Form.

COMMON *list-of: variable-reference*

where: *variable-reference* is: *variable-name*

or: *array-variable-name* ()

Note.

When a CHAIN, CHAIN MERGE or COMMON RESET command is executed all COMMON statements in the program at that time are found and obeyed. (This is done before any program lines are deleted!) At all other times COMMON statements are ignored.

Executing a COMMON statement has no effect.

Associated Keywords.

CHAIN, CHAIN MERGE, COMMON RESET

COMMON RESET

Remove all variables which are not COMMON. Command

Use.

COMMON RESET is similar to CLEAR except that variables which appear in COMMON statements are not removed.

Form.

COMMON RESET

Notes.

When a COMMON RESET is obeyed all variables which are not named in COMMON statements in the current program are removed.

COMMON RESET causes BASIC to forget any active FOR, WHILE or GOSUB commands.

Associated Keywords.

CHAIN, CHAIN MERGE, CLEAR, COMMON

CONSOLIDATE

Mark keyed file as consistent.

Jetsam Function

Use.

Once a keyed file has been changed it is marked inconsistent. This command causes all outstanding information to be written to disc and the file is marked consistent.

Form.

CONSOLIDATE (*file-reference*)

The *file-reference* must give the file number of an open keyed file.

Notes.

CONSOLIDATE returns an Integer value:

- On single user systems the value = 0
- On multiuser systems the value = the number of other users who have the file marked inconsistent.

The action of CONSOLIDATE is similar to CLOSEing and OPENing the keyed file, except that CONSOLIDATE retains any outstanding locks.

Associated Keywords.

CLOSE, OPEN

CONT

Continue after Break, STOP or END.

Command

Unavailable in Run Only Versions

Use.

When a program has been halted by Control-C (Break) or by a STOP or END command, the CONT command will resume program execution.

Form.

CONT

Notes.

CONT will be rejected if the program has been altered in any way since it halted.

Cosine.

Function

Use.

To calculate the Cosine of a given value, given that the argument is expressed in radians.

Form.

`COS (numeric-expression)`

The *numeric-expression* must yield an angle in the approximate range $-200,000 \dots 200,000$ radians.

Notes.

With angles very much greater than 2π the accuracy of this function becomes increasingly badly affected by the scaling of the angle into the range $-\pi \dots +\pi$. Rather than return a very inaccurate figure, BASIC will not evaluate COS for values much beyond the range given above.

If the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. COS returns a Single Length result.

Associated Keywords.

SIN, TAN, ATN

CREATE

Create a keyed data file.

Jetsam Command

Use.

To create and open a keyed file. Both the index and data files are created, initialised and opened.

Form.

```
CREATE [NEW] file-reference, data-filename, index-filename,  
                                           lock[, record-length]  
or CREATE [NEW] file-reference, data-filename, index-filename,  
                                           lock[, record-length], user-string
```

The *file-reference* gives the file number on which to open the keyed file once it has been created.

The *data-filename* is a *file-name-expression* giving the name of the data file to be created.

The *index-filename* is a *file-name-expression* giving the name of the index file to be created.

The *lock* specified is applied to the keyed file when it is opened.

The *record-length* is an *integer-expression* specifying the length of the records in the data file. The length must be in the range 1..*max* where *max* is set by the MEMORY and CLEAR commands, and when BASIC is first loaded (default value 128). If no value is specified 128 is assumed. The first two bytes of every record are reserved for use by Jetsam.

The *user-string* is a *string-expression* whose value is saved in the index file. The string may be any length, but if it is longer than 64 bytes it will be truncated. Jetsam does not attach any significance to this string, but the program may do.

Notes.

The files are opened and locked immediately they are created. The current position will be unset. It is an error (error 58) for either file to exist already. If the creation of one of the files succeeds but the other fails then CREATE will attempt to delete the one file created. One file without the other is a waste of space. (CREATE NEW will overwrite an existing file.)

When a file is created all ranks are set to allow multiple keys of the same value. RANKSPEC may be used to mark individual ranks to reject attempts to add new keys with the same value as existing keys.

Associated Keywords.

OPEN, CLOSE, RANKSPEC

Convert value to Single Length.

Function

Use.

To convert the value of the given expression to Single Length representation.

Form.

CSNG (*numeric-expression*)

Associated Keywords.

CINT, CDBL, MKS\$, CVS

Convert string to double length numeric.

Function

Use.

To reconvert a string produced by MKD\$, to recover the original Double Length value.

Form.

CVD (*string-expression*)

where the *string-expression* must yield a string eight bytes long.

Notes.

MKD\$ creates an eight byte string, which is an exact representation of a Double Length value, requiring no binary to decimal conversion. The string may be used, for instance, to put Double Length values into a random record.

Associated Keywords.

MKD\$, CVI, CVS

Convert string to integer.

Function

Use.

To reconvert a string produced by MKI\$, to recover the original Integer value.

Form.

CVI (*string-expression*)

where the *string-expression* must yield a string two bytes long.

Notes.

MKI\$ creates a two byte string, which is a representation for an Integer value requiring no binary to decimal conversion. The string may be used, for instance, to put Integer values into a random record.

Associated Keywords.

MKI\$, CVS, CVD

Convert string to integer.

Function

Use.

To reconvert a string produced by MKIK\$, to recover the original Integer value.

Form.

CVIK (*string-expression*)

where the *string-expression* must yield a string two bytes long.

Notes.

MKIK\$ creates a two byte string, which is a representation for an Integer value requiring no binary to decimal conversion and which is suitable for use as a Jetsam key.

Associated Keywords.

MKIK\$, MKUK\$, CVUK

Convert string to single length number.

Function

Use.

To reconvert a string produced by MKS\$, to recover the original Single Length value.

Form.

CVS (*string-expression*)

where the *string-expression* must yield a string four bytes long.

Notes.

MKS\$ creates a four byte string, which is an exact representation of a Single Length value, requiring no binary to decimal conversion. The string may be used, for instance, to put Single Length values into a random record.

Associated Keywords.

MKS\$, CVI, CVD

Convert string to single length integer.

Function

Use.

To reconvert a string produced by MKUK\$, to recover the original Integer numeric value.

Form.

CVUK (*string-expression*)

where the *string-expression* must yield a string two bytes long.

Notes.

MKUK\$ creates a two byte string, which is a representation for an Integer numeric value requiring no binary to decimal conversion and which is suitable for use as a Jetsam key.

Associated Keywords.

MKUK\$, MKIK\$, CVIK

Convert string to single length number.

Function

Use.

To reconvert a string produced by MKS\$, to recover the original Single Length value.

Form.

CVS (*string-expression*)

where the *string-expression* must yield a string four bytes long.

Notes.

MKS\$ creates a four byte string, which is an exact representation of a Single Length value, requiring no binary to decimal conversion. The string may be used, for instance, to put Single Length values into a random record.

Associated Keywords.

MKS\$, CVI, CVD

Convert string to single length integer.

Function

Use.

To reconvert a string produced by MKUK\$, to recover the original Integer numeric value.

Form.

CVUK (*string-expression*)

where the *string-expression* must yield a string two bytes long.

Notes.

MKUK\$ creates a two byte string, which is a representation for an Integer numeric value requiring no binary to decimal conversion and which is suitable for use as a Jetsam key.

Associated Keywords.

MKUK\$, MKIK\$, CVIK

DATA

Declare constant data.

Command

Form.

DATA *list-of: constant*

constant may be any form of *numeric-constant*, a *quoted-string* or an *unquoted-string*.

a *numeric-constant* may be surrounded by *white-space*, which is ignored.

a *quoted-string* may be surrounded by *white-space* which is ignored. The trailing double quotes may be omitted if the string is the last object on the line.

an *unquoted-string* is between 0 and 255 characters which are taken to be a string. A comma, colon or carriage return terminates *unquoted-strings* in this context. An *unquoted-string* may be surrounded by *white-space*, which is ignored.

Notes.

All the DATA commands in line number order within a program form a list of constants which may be read into variables by the READ command. As each constant is read a pointer is advanced to the next one. The RESTORE command moves the pointer to a specified position in this list.

DATA commands are not executed. The constants are not checked for validity until they are READ. Numeric constants may only be read into numeric variables. Strings (of either type) may only be read into string variables – noting that numeric constants will be treated as *unquoted-strings*.

Associated Keywords.

READ, RESTORE

Generate formatted string representation of number.

Function

Use.

To create a string representation of the value of a given expression, according to a given format template.

Form.

DEC\$ (*numeric-expression*, *format-template*)

The value of the *numeric-expression* is converted to a decimal string given the *format-template*, which is a *string-expression*, which is interpreted as in PRINT USING.

Notes.

The format template is not a full PRINT USING string, and may only contain the following ASCII characters:

+ - \$ * # , . ^

See Section 6.5.3 for a description of format specifications suitable for numbers.

Associated Keywords.

STR\$, HEX\$, OCT\$, PRINT USING

Define User Function.

Command

Illegal in Direct Mode

Use.

BASIC allows the program to define and use simple value returning functions. DEF FN is the definition part of this mechanism.

Form.

`DEF FNname[(formal-parameters)] = general-expression`

The *name* takes the same form as a *variable-name* (including an optional *type-marker*), the name of the function being *FNname*. The value returned by the function is of the same type as *name*.

The *formal-parameters* take the form *list-of: simple-variable-name*.

The *general-expression* may involve not only the formal parameters, but also other variables or functions. The result of the expression must be compatible with the type of the function.

Notes.

A user function is invoked when it is used as an argument in an expression. The invocation takes the form:

`FNname[(actual-parameters)]`

The *actual-parameters* take the form *list-of: general-expression*. There must be the same number of actual parameters as formal parameters. The type of each actual parameter must be compatible with the type of the corresponding formal parameter.

The formal parameters are local to the function. Variables with the same names as formal parameters are not affected by invoking the function, nor are they accessible from within the function.

If the type of the function is not explicit, then the default when the DEF FN is executed applies. If the type of a formal parameter is not explicit, then the default when the function is invoked applies.

DEF SEG

Define Segment Base Address.

Command

Use.

For Mallard-86 this command sets the Segment Base used in PEEK, POKE, USR, CALL, OPTION INPUT, OPTION LPRINT and OPTION PRINT.

Mallard-80 ignores this command.

Form.

DEF SEG[=*address-expression*]

Notes.

The default value of the User Segment Base is the base of BASIC's Data Segment (in which the user's program and data reside).

DEF SEG sets the User Segment Base to the value given. If the argument is omitted, then the User Segment Base is reset to its default value.

The address referenced in PEEK and POKE commands is given by the offset in the command and the User Segment Base most recently defined by DEF SEG.

The addresses of USR functions and CALLED subroutines are given by the offset in the command and the User Segment Base most recently defined by DEF SEG.

OPTION INPUT, OPTION LPRINT and OPTION PRINT all have an address parameter, specifying the address of a machine code subroutine. The segment base part of the full address of the subroutine is given by the User Segment Base at the time the OPTION INPUT/LPRINT/PRINT command was executed.

Associated Keywords.

CALL, USR, DEF USR, OPTION INPUT, OPTION LPRINT, OPTION PRINT, PEEK, POKE

Define external function.

Command

Use.

Provides an alternate mechanism to CALL for invoking external function subroutines. This mechanism allows for only one parameter, but allows the external function to return an anonymous value.

Form.

DEF USR[*digit*]=*address-expression*

Ten separate functions may be defined, USR0 to USR9. If the digit is omitted, 0 is assumed.

The *address-expression* gives the address of the external function.

Notes.

An external function is invoked when it is used as an argument in an expression, taking the form:

USR[*digit*] (*general-expression*)

where the *digit* may be one of 0...9; if omitted then 0 is assumed. There is only one parameter, which is mandatory.

USR[*digit*]'s may be redefined at will.

See Appendix III for details of external subroutines in general, and restrictions and caveats on external functions in particular.

In Mallard-86 the address specified gives the offset part of the full address. When the function is invoked the segment part is given by the User Segment Base most recently defined by DEF SEG.

Associated Keywords.

CALL, VARPTR, DEF SEG

DEFINT DEFSNG DEFDBL DEFSTR

Set default type for names.

Command

Use.

All variable and function names have a type associated with them. This type may be explicit in the name, or implicit, depending on the defaults set by these commands.

Form.

DEFINT *list-of: letter-range*

DEFSNG *list-of: letter-range*

DEFDBL *list-of: letter-range*

DEFSTR *list-of: letter-range*

where *letter-range* is: *letter*
or: *letter-letter*

where the form *letter-letter* defines an inclusive range of letters.

Notes.

When a variable name without an explicit *type-marker* is met the current default type is assumed. The default type depends on the first character of the name. The default is set for each letter by the latest of these commands in whose *letter-range-list* the letter was included.

DEFINT sets the default to Integer.

DEFSNG sets the default to Single Length.

DEFDBL sets the default to Double Length.

DEFSTR sets the default to String.

The initial state of the default type for all letters is Single Length. The state is reset to this by:

LOAD, RUN, CHAIN, NEW & CLEAR

DEL

Delete files.

MS-DOS (PC-DOS) Command

Use.

To erase a file or list of files.

Form.

DEL *list-of: file-name*

Notes.

This command is only available in versions of BASIC for MS-DOS (PC-DOS) type operating systems.

Each file specified is erased. If a file name contains 'wild card' characters, then all matching files are erased.

DEL takes the rest of the current line as its argument, irrespective of colons or single quotes.

Erasing an open file is not recommended. On multiuser systems this command may fail if the file is in use by other users or if the file is locked.

Associated Keywords.

ERA, KILL

DELETE

Delete lines of program.

Command

Use.

To remove part of the current program.

Form.

DELETE *line-number-range* [, *line-number*]

Notes.

Removes the lines in the given range, and has the following side effects:

- All User Functions are forgotten.
- ON ERROR GOTO is turned off.
- RESTORE action is taken.
- All active FOR, WHILE and GOSUB commands are forgotten.

If the *line-number* parameter is omitted BASIC returns to Direct Mode after executing a DELETE, otherwise BASIC starts execution at the given line. If the *line-number* given is zero, BASIC starts execution at the lowest numbered line.

Associated Keywords.

NEW

Delete a key.

Jetsam Function

Use.

To delete a key entry from the index file. If the key entry was the only one referring to the corresponding data record then the data record is automatically deleted. The current position is set to the next key after the deleted one.

Form.

`DELKEY (file-reference, lock[, index-position])`

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies what lock to apply to the record referred to by the key entry immediately following the entry to be deleted.

The *index-position* specifies the key entry to be deleted. If omitted the current position is assumed.

Return Codes.

- | | | |
|-----|---------|---|
| 0 | Success | the new current position is in the same key set as the deleted key (ie. it has the same key value as the deleted entry). |
| 101 | Success | the new current position is not in the same key set, but is in the same rank, as the deleted key (ie. it has a different key value from the deleted entry). |
| 102 | Success | the new current position is in the next, existing, rank. |
| 103 | Success | the deleted entry was the last entry in the index, the current position is unset. |
| 105 | Failure | could not find the specified or current position. The current position is set to the first key entry in the key set which would immediately follow the specified key if it were in the index. |
| 115 | Failure | the <i>index-position</i> parameter was omitted and the current position is unset. |
| 131 | Failure | could not write lock the record of the key to be deleted. |
| 132 | Failure | could not read lock the next record as requested. |
| 133 | Failure | could not write lock the next record as requested. |

Notes.

If the record referred to by the key entry to be deleted is not write locked, then DELKEY automatically obtains a temporary write lock. If this write lock cannot be obtained return code 131 is generated.

DELKEY may be used with ADDKEY to change a key's value. Since the record will be deleted by DELKEY if the record only has one key it is most important to perform the ADDKEY of the new key first.

To delete a record from the keyed file all its keys must be deleted. Once all a record's keys have been deleted it is no longer accessible, and the space it occupied will be re-used when a new record is added (using ADDREC).

Associated Keywords.

ADDKEY, ADDREC

DIM

Declare array dimensions.

Command

Use.

In order to allocate memory for an array BASIC must know how many dimensions it has, and the maximum permissible value of each one. The DIM command allows dimensions to be set explicitly.

Form.

DIM *list-of: subscripted-variable*

where *subscripted-variable* is: *variable-name (dimension-list)*

and *dimension-list* is: *list-of: integer-expression*

Notes.

Each *integer-expression* in the *dimension-list* sets the maximum value to be allowed for the corresponding subscript. The minimum value of each subscript is 0 or 1 as set in an OPTION BASE command, or 0 by default.

Array dimensions may be set explicitly in a DIM command, or implicitly when the array is first encountered. When an array is declared implicitly the maximum value for each subscript is set to 10.

It is an error to attempt to change the dimensions of an array which has already been declared (explicitly or implicitly).

There is no restriction on the number of subscripts an array may have, other than line length and store size!

Associated Keywords.

OPTION BASE

DIR

Directory Listing.

Command

Use.

To print a directory listing to the console. This may be a complete listing, or a partial listing depending on the given file names, which may contain 'wild card' characters.

Form.

DIR [*list-of: filename*]

Notes.

If the list of filenames is omitted, then all files on the default drive are listed. If a number of filenames is given, then each name is taken in turn and all files consistent with the name are listed. The 'wild card' characters '?' and '*' may be used in the usual way.

DIR treats all the rest of the current line as its argument, irrespective of colons or single quotes.

Associated Keywords.

FILES, FINDS

DISPLAY

Display contents of file on console.

Command

Use.

To print a given file on the console.

Form.

DISPLAY *string-expression*

The *string-expression* yields a string giving the name of the file to be printed on the console. The name may not contain 'wild card' characters.

Notes.

The file specified is read and written directly to the console. Typing Control-C will abandon the operation and return to Direct Mode (except in Run-Only versions, when Control-C is ignored).

While the file is being written to the console, typing Control-S will suspend output (except in Run-Only versions, when Control-S is ignored). Output may be restarted in the usual way (see Section 3.3).

Tabs and carriage returns are treated in the usual way, and console width is enforced. All other characters are sent to the console exactly as read from the file, so control characters and characters outside the normal ASCII range may be included in the file to produce special console effects.

Associated Keywords.

TYPE

EDIT

Edit a program line.

Command

Unavailable in Run Only Versions

Use.

To amend individual program lines.

Form.

`EDIT line-number`

Notes.

Enters the line editor with the given line as the line to be edited.

See Chapter 4 for a description of the line editor.

END

End of program.

Command

Use.

To signal the end of a program, and stop cleanly.

Form.

END

Notes.

Any number of END commands may appear anywhere in a program, though an END is assumed after BASIC has obeyed the last line of a program.

END closes all files and returns to Direct Mode.

Associated Keywords.

STOP

End of File test.

Function

Use.

To test if the given file is positioned at end of file. Returns -1 (true) if is at end of file. Returns 0 (false) otherwise.

Form.

EOF (*file-number-expression*)

Notes.

The file may not be open for Output and may not be a keyed file.

EOF is most useful with sequential input files. A WHILE NOT EOF loop may be used to read up to the end of a file.

EOF may be used with random files, but must be used with care. EOF does not return a valid result before the first GET operation following the OPEN for the file. Following a GET, EOF will return:

- false if the record has ever been written.
- true if the record read is beyond the highest numbered record ever written.

EOF may, therefore, be used to read sequentially a random file in which all records from the first up to the highest numbered have been written at some time. EOF will return true once the first record beyond the end of file has been read. A WHILE NOT EOF loop preceded by a GET for the first record, and with the GET for the next record at the end, would be suitable.

However, this technique must be used with care under CP/M and MS-DOS 1, which allocate disc space in units of 128 bytes. If your record size is not a multiple of 128 bytes, EOF may return FALSE spuriously at the end of the file because PUTting the highest numbered record in a file also wrote part of the next 128-byte unit (or units). Furthermore, if you leave some gaps, ie. you do not write to all the records in a file, CP/M may report EOF true for these gaps.

Associated Keywords.

LOC, LOF

ERA

Erase files.

CP/M Command

Use.

To erase a file or list of files.

Form.

ERA *list-of: file-name*

Note.

This command is only available in versions of BASIC for CP/M type operating systems.

Each file specified is erased. If a file name contains 'wild card' characters, then all matching files are erased.

ERA takes the rest of the current line as its argument, irrespective of colons or single quotes.

Erasing an open file is not recommended. On multiuser systems this command may fail if the file is in use by other users or if the file is locked.

Associated Keywords.

KILL

ERASE

Erase arrays.

Command

Use.

When an array is no longer required ERASE may be used to remove it and reclaim the memory it occupies ready for other use.

Form.

ERASE *list-of: variable-name*

where each *variable-name* is the name of an array, without any subscript part.

Notes.

It is an error to ERASE an array which has not been declared (explicitly or implicitly).

Once an array has been ERASEd it is possible to declare it again with new dimensions.

Associated Keywords.

DIM

Error Line Number.

Function

Use.

ERL may be used in error handling subroutines to discover the number of the line being executed in which the error occurred.

Notes.

This function may be used in expressions in the usual way.

Warning.

If ERL is used in comparison expressions it must be written on the lefthand side for RENUM to recognise the righthand side as a line number. If the expression is written the other way round, RENUM is unable to find the line number and will ignore it – which is likely to cause inconsistencies.

Associated Keywords.

ON ERROR, ERR, ERROR, OSERR

ERR

Error Number.

Function

Use.

ERR may be used in error handling subroutines to discover the number of the error which has occurred.

Notes.

This function may be used in expressions in the usual way.

Associated Keywords.

ON ERROR, ERL, ERROR, OSERR

ERROR

Cause Error action to be taken.

Command

Use.

Invoke Error Action with a given error number.

Form.

ERROR *integer-expression*

where the *integer-expression* must evaluate to a number in the range 1...255.

Notes.

If the error number is one already used and recognised by BASIC, the action taken is the same as would be taken if such an error had been detected by BASIC.

Error numbers beyond those recognised by BASIC may be used by the program to signal its own errors. Error numbers 200 or greater are not used by BASIC and will not be used in future versions.

In the event of BASIC attempting to print an error message for an error number that it does not recognise, the message 'Unknown error' is produced.

Associated Keywords.

ON ERROR, ERR, ERL

EXP

Exponential.

Function

Use.

Calculate e to the given power, where e is the number whose natural logarithm is 1 (approx. 2.7182818).

Form.

EXP (*numeric-expression*)

Notes.

EXP of numbers much greater than 88 will cause an overflow error.

EXP of numbers much less than -88.7 underflows, and yields the value zero.

While the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. EXP returns a Single Length result.

Associated Keywords.

LOG

FETCHKEY\$

Fetch the current key value.

Jetsam Function

Use.

Fetch the key value from the key entry at the current position.

Form.

FETCHKEY\$ (*file-reference*)

The *file-reference* gives the file number of an open keyed file.

Notes.

This function returns the key value of the last key entry found. Unless the associated record, or the file, is locked there is no guarantee that this key entry still exists.

Associated Keywords.

FETCHRANK, FETCHREC

FETCHRANK

Fetch the current key rank.

Jetsam Function

Use.

Fetch the rank from the key entry at the current position.

Form.

FETCHRANK (*file-reference*)

The *file-reference* gives the file number of an open keyed file.

Notes.

This function returns the rank from the last key entry found. Unless the associated record, or the file, is locked there is no guarantee that this key entry still exists.

Associated Keywords.

FETCHKEY, FETCHREC

FETCHREC

Fetch the current record number.

Jetsam Function

Use.

Fetch the record number of the record referred to by the key entry at the current position.

Form.

FETCHREC (*file-reference*)

The *file-reference* gives the file number of an open keyed file.

Notes.

If the current position is set this function returns the record number from the key entry. Unless the record, or the file, is locked there is no guarantee that this key entry still exists.

If the current position is unset, this function returns zero.

Associated Keywords.

FETCHRANK, FETCHKEY\$

FIELD

Define a field layout for a given file.

Command

Use.

A FIELD command defines a template for records in a file open for random access, and associates a string variable with each field.

Form.

FIELD *file-reference*, list-of: *field*

where:

The *file-reference* gives the file number to which this template applies. The file must be open for random access.

field is: *field-size* AS *string-variable*

where *field-size* is an *integer-expression*, value in the range 0...255. The sum of all the *field-sizes* must be less than or equal to the record size.

Notes.

Each field in the template is given a size and a string variable is associated with it. The fields are mapped onto the record, with the first field corresponding to the first few bytes, the second the following bytes, and so on.

The string variables are effectively pointers into the record buffer. Executing a FIELD command neither reads nor writes data, but simply sets the string variables to point at the current contents of the buffer.

Each FIELD statement executed for a given file sets up a separate template for the record buffer. There is no restriction on the number of such templates, and they will all be valid.

The number of bytes available to the FIELD command in a keyed file is two bytes less than the record length. These reserved bytes cannot be accessed by the user or be defined as part of the FIELD template.

Warning.

The string variable associated with each field must not be assigned to except by using LSET, RSET or MID\$. Any other form of assignment will not write to the record buffer, but will create a new string which will not be stored in the random record buffer and will break the association of the string variable name with the record buffer.

Associated Keywords.

OPEN, GET, PUT, LSET, RSET, MID\$

Directory listing.

Command

Use.

To print a directory listing to the console. This may be a complete listing, or a partial listing depending on a given file name, which may contain 'wild card' characters.

Form.

`FILES [file-name-expression]`

The *file-name-expression* specifies which files are to be listed. If the expression is omitted, then all files on the default drive are listed.

Notes.

The 'wild card' characters '?' and '*' may be used in the string expression to specify a number of files in the usual way.

Associated Keywords.

DIR, FIND\$

FIND\$

Look for given file.

Function

Use.

FIND\$ looks up the given file and returns its name, if found.

Form.

FIND\$ (*file-name-expression*[, *ordinal*])

The *file-name-expression* gives the name of the file to be searched for. The resulting string may contain 'wild card' characters, which are treated in the usual way.

The *ordinal*, if present, must be an *integer-expression*.

Notes.

If the file sought is found FIND\$ returns a twelve character string containing the name of the file, complete with any file attribute bits. (Note that the STRIP\$ function may be used to remove these attribute bits.)

If no matching file is found, then a null string is returned.

The *ordinal* argument is useful when the *file-name-expression* contains 'wild card' characters. If the value of the *ordinal* is *n*, then FIND\$ looks for the *n*th matching file. If the *ordinal* is omitted then FIND\$ looks for the first matching file.

Associated Keywords.

DIR, FILES, UPPER\$, STRIP\$

FINDDIR\$

Look for given directory.

Function

Use.

For Mallard-86 MSDOS 2 version, this function looks for the given directory and returns its name, if found.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all return a null string.

Form.

FINDDIR\$ (*string-expression* [, *ordinal*])

where the *string-expression* yields a *file-spec* which gives the name of the directory to search for. The *file-simple* may contain 'wild-cards'.

The *ordinal*, if present, must be an *integer-expression* yielding a value in the range 1...32767.

Notes.

If the directory sought is found, FINDDIR\$ returns a twelve character string containing the name, *dir-simple*, of the directory.

If no matching directory is found then a null string is returned.

The *ordinal* argument is useful when the *string-expression* contains 'wild-cards'. If the value of the *ordinal* is *n*, then FINDDIR\$ looks for the *n*th matching directory. If the *ordinal* is omitted then 1 is assumed.

The associated function FIND\$ cannot be used for searching for directories.

Associated Keywords.

CHDIR, MKDIR, RMDIR, CD, MD, CHDIR\$, FIND\$

FIX

Fix number to integer.

Function

Use.

To truncate given value to an integer. Note that this does not convert floating point numbers to integer format, but simply removes any fraction part, rounding the value toward zero.

Form.

FIX (numeric-expression)

Associated Keywords.

CINT, INT, ROUND

FOR

FOR loop.

Command

Use.

Execute a section of a program a given number of times, stepping a control variable between a start and an end value.

Form.

FOR *simple-variable*=*start* TO *end* [STEP *step-size*]

simple-variable is the control variable for the FOR loop, it must be an Integer or Single Length, and may not be an array item.

start is a *numeric-expression* giving the initial value for the control variable.

end is a *numeric-expression* giving the limiting value for the control variable.

step-size is a *numeric-expression* giving the value to add to the control variable after each iteration. If the step size is omitted, a step of 1 is assumed. Note that the *step-size* may be negative.

The results of the *start*, *end* and *step-size-expressions* are forced to the same type as the control variable.

Notes.

The FOR command initiates a FOR loop. The end of the FOR loop is marked by a NEXT command.

When the FOR command is executed the *start*, *end* and *step-size* expressions are evaluated, then the control variable is set to the *start* value. If the initial value is beyond the end value (see below) then the FOR loop is skipped, otherwise the loop is entered. When the matching NEXT is found the step size is added to the control variable, and control returns to the start of the FOR loop unless the control variable is now beyond the end value.

If the step size is positive (or omitted) the FOR loop terminates when the control variable is greater than the end value. If the step size is negative the FOR loop terminates when the control variable is less than the end value.

The NEXT command which matches a given FOR is established statically when the FOR is first executed. That is to say that the NEXT which matches the FOR depends on the order of statements in the program, quite independently of the order of execution. It is not possible, therefore, to have more than one NEXT associated with a given FOR.

FOR loops may be nested, both inside other FOR loops and inside WHILE loops.

The value of the control variable is defined once the loop has terminated. It is permissible to terminate a FOR loop by avoiding the NEXT; the value of the control variable is unchanged.

Care must be exercised if fractional step sizes are used, since the effects of rounding may mean that the control variable value can never exactly equal the final value.

Associated Keywords.

NEXT, WHILE

Free space measurement.

Function

Use.

To establish how much free memory remains unused by BASIC. Two forms of the function exist, one causes BASIC to 'garbage collect' before measuring the free space.

Form.

FRE (*numeric-expression*)

or FRE (*string-expression*)

Notes.

The value of the argument in a FRE is not relevant, so FRE (0) and FRE ("") are the recommended forms.

BASIC maintains a heap in the free memory, in which strings are stored. As the heap grows the free space is used up. At the same time areas within the heap area may become free, but cannot be reclaimed by BASIC until a 'garbage collection' is performed. In general BASIC does not 'garbage collect' until the heap has grown to fill all free memory.

FRE (0) returns the amount of memory left for the heap to grow into.

FRE ("") forces BASIC to 'garbage collect' so that any unused space within the heap is returned to the free area. The result of the expression is then the maximum free space available. Note that if there are a large number of strings the 'garbage collection' may take a noticeable time.

Associated Keywords.

MEMORY, CLEAR, HIMEM

GET

Get record from random or keyed file.

Command

Use.

To read a given record from a random file or a keyed data file into the record buffer, where it can be processed by the program.

Form.

GET *file-reference*[, *record-number*]
or GET *file-reference*, [*record-number*], *lock*

The *file-reference* specifies the file to read from, which must be an open random or keyed file.

For random files: The *record-number* specifies the record to be read. If no number is specified the record following the one in the last GET or PUT is assumed – if this is the first GET or PUT then record 1 is read.

The *lock* parameter, if present, specifies the record lock to be applied to the record before it is read. This parameter may only be present when reading from random files.

For keyed files: The *record-number* specifies which record from the data file is to be read. If the parameter is omitted the record number given by the current position is used.

For Random Files Note.

An internal pointer is associated with the random record buffer, to allow INPUT # et al to read data from the record. A GET operation resets that pointer.

For Keyed Files Note.

The record, or the file, must be read or write locked. Only record numbers that have been returned by the FETCHREC function should be used.

For Random Files in Multiuser Versions Note.

GET releases any outstanding temporary record write lock. See Section 9.3.4 for a description of the temporary record write lock mechanism.

Record locking is only required if the file is open unlocked. If a *lock* is specified then BASIC attempts to gain the lock before reading the record and will report an error 69 if the lock cannot be obtained. The record may be locked using the LOCK function before the GET is issued. If the record is not locked when GET comes to read it BASIC attempts to gain a temporary record write lock.

Associated Keywords.

OPEN, FIELD, INPUT #, LINE INPUT #, PUT

GOSUB

Go to Subroutine.

Command

Use.

To call a given subroutine.

Form.

GOSUB *line-number*
or GO SUB *line-number*

Notes.

When GOSUB is executed the program jumps to the given line number, remembering the position immediately after the GOSUB so that execution may continue there when the subroutine returns.

Subroutines are terminated by a RETURN command. A subroutine may contain more than one RETURN command.

Subroutines may be nested and recurse, to a depth limited only by the stack size.

Associated Keywords.

RETURN, ON *expression* GOSUB

GOTO

Go to line.

Command

Use.

Unconditional jump to a given line.

Form.

GOTO *line-number*

or GO TO *line-number*

Associated Keywords.

ON *expression* GOTO

*

Hexadecimal string.

Function

Use.

To produce a string of hexadecimal digits representing the value of the given expression.

Form.

HEX\$ (*unsigned-integer-expression* [, *field-size*])

The value of the *unsigned-integer-expression* is treated as an unsigned 16 bit integer, to be converted into a string of hexadecimal characters. See Section 2.10 for a description of unsigned integer.

The optional *field-size* is an *integer-expression* which gives the minimum size of string to be produced, and must yield a value in the range 0...16.

Notes.

HEX\$ always generates as many characters as are required to represent the number (zero generates at least one digit). If the optional *field-size* is present and the string is too short, then the string is filled to length with leading zeros. The string is not truncated if it is too long.

Associated Keywords.

OCT\$, DEC\$, STR\$

HIMEM

Return address of highest byte in memory used by BASIC.

Function

Use.

The amount of memory used by BASIC may be altered by the MEMORY and CLEAR commands. The built-in HIMEM function may be used to establish the current memory use.

Form.

HIMEM

Notes.

HIMEM returns a Single Length value giving the address of the highest byte in memory used by BASIC.

Associated Keywords.

CLEAR, MEMORY, FRE

Use.

To conditionally execute statements.

Form:

IF *logical-expression* THEN *option-part* [ELSE *option-part*]
or IF *logical-expression* GOTO *line-number* [ELSE *option-part*]

where *option-part* is: *statement(s)*
or: *line-number*

and *statement(s)* is one or more statements on the same line as the IF (separated by colons).

Notes.

The *logical-expression* is evaluated and if the result is non-zero then the THEN or GOTO option is executed. Otherwise the THEN or GOTO option is skipped, until either end of line or a matching ELSE is found. In the latter case the ELSE option is executed.

IF statements may be nested to any depth, limited only by the line length. ELSE parts are associated with the innermost IF which does not already have an ELSE part.

THEN *line-number* and ELSE *line-number* are equivalent to THEN GOTO *line-number* and ELSE GOTO *line-number*, respectively.

The IF command is terminated by end of line. It is not possible to have further statements independent of the IF later on the same line.

Associated Keywords.

WHILE

INKEY\$

Input key from keyboard.

Function

Use.

Input next key, if any, from keyboard (console input). Note that any character input is not reflected to the console output.

Form.

INKEY\$

Notes.

If there is a character pending then INKEY\$ returns it (as a one character string). If no characters are pending INKEY\$ returns an empty string.

Except for Control-C and Control-S, all characters are passed as read to the program, without reflecting them to the console. Control-C halts the program in the usual way. Control-S suspends the execution of the program in the usual way. It is not possible, therefore, for a program to give anomalous results depending on the exact moment that these keys are pressed.

OPTION RUN modifies the handling of Control-C and Control-S. While OPTION RUN is in force all versions of BASIC treat these characters as ordinary characters, and INKEY\$ will pass them to the program.

To be consistent with Full versions, Run Only versions usually ignore both Control-C and Control-S, but will pass them to the program if OPTION RUN is in force.

Associated Keywords.

INPUT\$, OPTION RUN, OPTION STOP

Input from I/O port.

Function

Use.

Input value from the given I/O port.

Form.

INP (*port-number*)

INPW (*port-number*)

For Mallard-80 the *port-number* is an *integer-expression* which must yield a value in the range 0...255.

For Mallard-86 the *port-number* is an *address-expression*

Notes.

INP reads a byte from the given I/O port.

INPW reads a word from the given I/O port. INPW is illegal in Mallard-80.

Associated Keywords.

OUT, OUTW, WAIT, WAITW

INPUT

Input data from console.

Command

Use.

To prompt the operator and read data from the console.

Form.

INPUT [;][*quoted-string* ;] *list-of:variable*

or INPUT [;][*quoted-string* ,] *list-of:variable*

where the *quoted-string* is an optional prompt string.

Notes.

The default prompt is a question mark. If an explicit prompt string is given then a question mark is appended to it if the first form is used (with a semicolon after the prompt string), but not if the second form is used.

When the INPUT command is executed the prompt is issued, and a line is read from the console. The line input is treated as a list of items, and BASIC attempts to assign the value of one item to each variable in the list in turn, checking that the item is compatible with the variable. The line read takes the form:

list-of: item

where each *item* may be one of:

a *numeric-value*, which may be surrounded by *white-space*, which is ignored.

a *quoted-string*, which may be surrounded by *white-space*, which is ignored. The trailing double quotes may be omitted if the string is the last object on the line.

an *unquoted-string*, which may be surrounded by *white-space*, which is ignored. An *unquoted-string* is between 0 and 255 characters terminated by comma or carriage return.

If there are the right number of items on the line, and their types are all compatible with the corresponding variable, then the values of the items are assigned to the variables.

If there are too few or too many items, or if an item is not compatible with the corresponding variable, then the message '?Redo from start' is issued, and the INPUT command restarted (so the prompt is repeated). None of the *variables* in the INPUT statement will have been affected.

The optional semicolon immediately after the INPUT keyword stops BASIC from reflecting the carriage return typed at the end of the input line. This means that the cursor is left at the end of the text just entered.

When fetching a line for INPUT, Simple Line Input is used (see Section 4.1).

Values are assigned to the *variables* in the order they appear in the INPUT statement. A *variable* may be an item in an array. If any variable which appears earlier in the INPUT statement also appears in a subscript expression, then the newly assigned value will be used when establishing the array item to be assigned to. (If this feature is to be exploited the subscript expression must evaluate to a valid subscript both before and after the variables in question have been affected by the INPUT.)

Associated Keywords.

LINE INPUT, DATA, READ, INPUT #, LINE INPUT #, INKEY\$, INPUT\$

INPUT

Input data from file.

Command

Use.

To read data from sequential files, or the random record buffer.

Form.

INPUT #*file-number-expression*, *list-of: variable*

The *file-number-expression* specifies the file to read from. If the file is open for random access, then the data is read from the record buffer.

The *list-of: variable* specifies where to read data to and what type of data is to be read.

Notes.

BASIC attempts to read one item from the file for each variable in the list of variables. Each item must be compatible with the variable to which its value is to be assigned. Each item in the file may be one of:

a *numeric-value*, which may be preceded by *white-space* which is ignored. A numeric item is terminated by *white-space*, comma, carriage return or end of file. Trailing *white-space* is ignored. Any following comma or carriage return is ignored.

a *quoted-string*, which may be preceded by *white-space*, which is ignored. All characters after the leading double quotes, including carriage returns and linefeeds, up to the trailing double quotes form the string. The string is terminated by double quotes or end of file. After the trailing double quotes, trailing *white-space* is ignored. Any following comma or carriage return is ignored.

an *unquoted-string*, which may be preceded by *white-space*, which is ignored. An *unquoted-string* item is terminated by a comma, carriage return or end of file.

(Nulls are ignored throughout. Carriage return immediately followed by Line Feed is treated as a carriage return, and vice versa. Strings are limited to 255 characters, and terminate automatically after the 255th.)

If the file is open for random access, then INPUT # reads from the record buffer, and will fail (Error 50) if an attempt is made to read beyond the end of the buffer. An internal pointer is associated with the record, to allow separate INPUT # commands to read along the record. This pointer is reset to the beginning of the buffer each time a record is read (by GET).

Associated Keywords.

INPUT #, LINE INPUT, DATA, READ, LINE INPUT #, INKEY\$, INPUT\$, GET

INPUT\$

Input fixed length string.

Function

Use.

To input a fixed number of characters either from the keyboard or from a file.

Form.

INPUT\$ (*integer-expression* [, [#]*file-number-expression*])

The *integer-expression* gives the number of characters to be read, and must yield a value in the range 1...255.

The *file-number-expression* gives the number of the file from which to read. The file must be open for Random or Input Access. If the file number part is omitted, INPUT\$ reads from the console.

Notes.

When reading from the console all characters, except for Control-C and Control-S, are included in the string as read, without reflecting them to the console. Control-C halts the program in the usual way. Control-S suspends the execution of the program in the usual way. It is not possible, therefore, for a program to give anomalous results depending on the exact moment that these keys are pressed.

OPTION RUN modifies the handling of Control-C and Control-S. While OPTION RUN is in force, BASIC treats these as ordinary characters, and INPUT\$ will pass them to the program.

When reading from a file open for Random Access it is an error to attempt to read beyond the end of the current record.

When reading from a random access or keyed file no special notice is taken of the character Control-Z.

To be consistent with Full versions Run Only versions usually ignore both Control-C and Control-S, but will pass them to the program if OPTION RUN is in force.

Associated Keywords.

INKEY\$, INPUT #, OPTION RUN, OPTION STOP

Search for substring in string.

Function

Use.

To search from a given point in a given string for the first occurrence of another given string.

Form.

INSTR (*[integer-expression*, *]searched-string*, *searched-for-string*)

The *integer-expression* gives the character position within the *searched-string* at which to start searching, and must yield a value in the range 1...255. If omitted then the search starts at the first character of the searched string.

The *searched-string* is a *string-expression* giving the string to be searched along.

The *searched-for-string* is a *string-expression* giving the string to search for.

Notes.

If the *searched-for-string* is found, INSTR returns the position within the *searched-string* of the start of the first occurrence of the *searched-for-string*. Otherwise INSTR returns zero.

If there are no characters in the *searched-string* from the given (or default) starting position, then INSTR always returns zero.

If the *searched-for-string* is null, then it is immediately found, unless the previous rule applies.

INT

Number to integer.

Function

Use.

To round a given value to the nearest smaller integer (round toward minus infinity). Note that this does not convert floating point numbers to integer format, but simply removes any fraction part.

(For positive numbers this is equivalent to FIX. For negative numbers the value returned is one less than FIX would return, unless the value was already integral.)

Form.

INT (*numeric-expression*)

Associated Keywords.

CINT, FIX, ROUND

KILL

Kill file.

Command

Use.

To erase a file.

Form.

KILL *file-name-expression*

The *file-name-expression* gives the name of the file to be erased.

Notes.

If the *file-name-expression* contains 'wild card' characters, then all matching files are erased.

Erasing an open file is not recommended. On multiuser systems this command may fail if the file is in use by other users or if the file is locked.

Associated Keywords.

ERA

LEFT\$

Extract lefthand part of string.

Function

Use.

To extract a given number of characters from the lefthand end of a given string.

Form.

LEFT\$ (*string-expression*, *integer-expression*)

The *string-expression* gives the string from which to extract characters.

The *integer-expression* gives how many characters to extract, and must yield a value in the range 0...255.

Notes.

If the given string is shorter than the required length, then the entire string is returned. Otherwise the lefthand end of the string is returned, giving a string of the required length.

Associated Keywords.

MID\$, RIGHT\$

LEN

Determine length of string.

Function

Use.

To determine the length of a given string.

Form.

LEN (*string-expression*)

Notes.

LEN returns an integer in the range 0...255. 0 indicates that the string is null.

All characters in the string are counted, including any non-printing ones.

LET

Preface an assignment.

Command

Use.

The LET command is not very useful, it is really a hangover from the earliest BASICs, and may be ignored.

Form.

LET *variable=expression*

Notes.

The expression is evaluated and the resulting value assigned to the variable, unless the two are incompatible.

The LET keyword is, in fact, wholly redundant (except in the Run Only Special Direct Mode).

LINE INPUT

Input complete line from console.

Command

Use.

To prompt the operator and read a complete line of text from the console to a given string variable.

Form.

LINE INPUT [;][*prompt*;*string-variable*
or LINE INPUT [;][*prompt*,*string-variable*

The *prompt*, if present, must be a *quoted-string*.

The *string-variable* gives the name of the variable to which to assign the string created.

Notes.

The two forms are equivalent.

When the LINE INPUT command is executed the *prompt*, if any, is issued, and a line is read from the console and assigned to the given string. The line is terminated by carriage return (or after 255 characters, whichever is the sooner).

If a semi-colon immediately follows the LINE INPUT then BASIC will not reflect the carriage return typed at the end of the input line. This means that the cursor is left at the end of the text just entered.

When fetching a line for LINE INPUT Simple Line Input is used (see Section 4.1).

Associated Keywords.

DATA, READ, INPUT, INPUT #, LINE INPUT #, INKEY\$, INPUT\$

LINE INPUT

Input complete line from file.

Command

Use.

To read a complete line from a given file to a given string.

Form.

LINE INPUT #*file-number-expression*, *string-variable*

Notes.

The file must be open for Input or Random Access.

If the file is open for random access, then LINE INPUT # reads from the record buffer, and will fail (Error 50) if an attempt is made to read beyond the end of the buffer. An internal pointer is associated with the record, to allow separate LINE INPUT # commands to read along the record. This pointer is reset to the beginning of the buffer each time a record is read (by GET).

Associated Keywords.

DATA, READ, INPUT, INPUT #, LINE INPUT, INKEY\$, INPUT\$

LIST LLIST

List program.

Command

Unavailable in Run Only versions

Use.

To list the current program. Part or all of the program may be listed. LIST sends its output to the console. LLIST sends its output to the line printer.

Form.

LIST *[line-number-range]*

LLIST *[line-number-range]*

The line number range defines which lines of the program to list. If it is omitted then all the program is listed.

Notes.

BASIC returns to Direct Mode after obeying a LIST or LLIST command.

Typing Control-C will abandon the listing and return to Direct Mode.

LOAD

Load a program into memory.

Command

Use.

To read a program from disc into memory, replacing any existing program. The program may optionally be run immediately.

Form.

LOAD *string-expression* [, R]

The *string-expression* gives the name of the file from which to read the program. If no file type extension is given .BAS is assumed.

Notes.

The optional , R indicates that the program is to be run immediately it is loaded.

Any existing program, user functions and variables are deleted from memory. DEFINT, DEFSNG, DEFDBL, DEFSTR and OPTION BASE settings are reset. Unless the , R option is specified all files are closed.

In Run Only versions the , R is required, otherwise BASIC will terminate and return to system level after loading the given program.

Associated Keywords.

SAVE, RUN, MERGE, CHAIN, CHAIN MERGE

LOC

Current Location in file.

Function

Use.

To establish the current record number in a given file.

Form.

LOC (*file-number-expression*)

where *file-number-expression* gives the number of a file which is open (for any form of access).

Notes.

If the file is open for Random Access LOC returns the record number of the record referenced in the last PUT or GET operation. If there have been no PUT or GET operations since the file was opened then 0 is returned.

If the file is open for Input or Output Access then LOC returns the number of 128 byte records read or written.

LOC may not be used with keyed files.

Associated Keywords.

LOF, EOF

LOCK

Change a record lock.

Jetsam Function

Use.

To change or clear a record lock.

Form.

LOCK (*file-reference*, *lock*, *record-number*)

The *file-reference* gives the file number of an open random or keyed file.

The *lock* parameter specifies what record lock is to be applied to the given record.

The *record-number* specifies which record from the data file is to have its lock changed.

Return codes.

0	Success	unqualified.
130	Failure	the file is read locked by another user.
132	Failure	could not read lock the record as requested.
133	Failure	could not write lock the record as requested.

Notes.

For keyed files only record numbers that have been returned by the FETCHREC function should be used. When the record was found, using one of the various SEEK functions, a record lock will have been applied to the record. The LOCK function allows the type of lock to be changed or released. The record must already be read or write locked.

For random files the LOCK function may be used at any time to set, change or release locks on any existing record in the file. The record need not already be locked. Record locks may also be affected by GET and PUT commands. The LOCK function interacts with the temporary record write lock mechanism – see Section 9.3.4.

Associated Keywords.

GET, PUT, SEEKRANK, SEEKKEY, SEEKREC, SEEKSET, SEEKNEXT, SEEKPREV

LOF

Length of file.

Function

Use.

To give some indication of the length of a given file.

Form.

LOF (*file-number-expression*)

where *file-number-expression* gives the number of a file which is open (for any form of access).

Notes.

The value returned gives some indication of the length of the file, the exact value returned depends on the underlying operating system. LOF will never return a zero value unless the file is empty.

LOF used with keyed files returns the length of the data file.

Associated Keywords.

LOC, EOF

LOG

Natural logarithm.

Function

Use.

To calculate natural logarithms.

Form.

LOG (*numeric-expression*)

where the result of the *numeric-expression* must be greater than zero.

Notes.

While the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. LOG returns a Single Length result.

Associated Keywords.

EXP, LOG10

LOG10

Logarithm, base 10.

Function

Use.

To calculate logarithms to base 10.

Form.

LOG10 (*numeric-expression*)

where the result of the *numeric-expression* must be greater than zero.

Notes.

While the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. LOG10 returns a Single Length result.

Associated Keywords.

EXP, LOG

LOWERS\$

Convert string to lower case.

Function

Use.

To create a new string which is a copy of another, with all upper case alphabetic characters converted to lower case equivalents.

Form.

LOWERS\$ (*string-expression*)

Notes.

The result of the *string-expression* is returned, with any characters in the range A . . . Z converted to the equivalent character in the range a . . . z.

Associated Keywords.

UPPERS\$

Line printer position.

Function

Use.

To establish the current print position on the line printer.

Form.

LPOS (*numeric-expression*)

where the *numeric-expression* is a dummy argument, so LPOS (0) is the recommended form.

Notes.

LPOS returns a logical position on the line printer, which may bear no relation whatsoever to the current position of the print head!

In calculating the logical position all non-printing characters (those with values < 32) are not counted, except for:

- Backspace (value 8), which counts as -1 unless the position is one.
- Tab (value 9), which expands to spaces, each of which is counted. (Tab positions eight characters apart all the way across the printer are implicitly defined.)
- Carriage return (value 13), which sets the position to one.

The position is also affected by carriage returns inserted by BASIC if the logical position exceeds the width of the printer, as set by WIDTH LPRINT.

If the width is set infinite (by WIDTH LPRINT 255) then the logical position will grow to 255, but no further.

Associated Keywords.

WIDTH LPRINT, POS

LPRINT

Print to the Line Printer.

Command

See PRINT, reading Line Printer for Console throughout.

Associated Keywords.

LPOS, PRINT

Set one string to another, left justified.

Command

Use.

To replace the contents of one string by another, filling with spaces on the right to the same length as the original contents.

Form.

LSET *string-variable* = *string-expression*

Notes.

The *string-expression* is evaluated and then forced to the same length as the current value of the given *string-variable* – either by padding with spaces at the righthand end, or by discarding characters from the righthand end. The result replaces the original contents of the *string-variable*.

LSET is particularly useful for setting new values into fields of a random record.

Associated Keywords.

FIELD, RSET, MID\$, MKI\$, MKS\$, MKD\$

MAX

Determine maximum value.

Function

Use.

To determine the maximum of a number of values.

Form.

MAX (*list-of: numeric-expression*)

Notes.

MAX returns the value of the largest of the *numeric-expression(s)*.

Associated Keywords.

MIN

Make a new directory.

Command

Use.

For Mallard-86 MSDOS 2 version, this makes a new directory on the specified, or default drive.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all ignore this command.

Form.

MD *dir-name*

where the last *dir-simple* of the *dir-name* specifies the directory to make.

Notes.

The directory must not already exist.

This command takes the rest of the current line as its argument, irrespective of colons or single quotes.

Associated Keywords.

CHDIR, RMDIR, MKDIR, CD, RD, CHDIR\$, FINDDIR\$

MEMORY

Reset BASIC memory parameters.

Command

Use.

To change the amount of memory used by BASIC, the maximum number of files and the maximum random record size.

Form.

MEMORY *high-memory*
or MEMORY [*high-memory*], *stack-size*
or MEMORY [*high-memory*], [*stack-size*], *number-of-files*
or MEMORY [*high-memory*], [*stack-size*], [*number-of-files*], *maximum-record-length*

High-memory is an *address-expression* which gives the address of the highest byte in memory which may be used by BASIC. If omitted the current setting is retained.

Stack-size is an *address-expression*, which gives the number of bytes BASIC should use for its stack. If omitted the current setting is retained. If present, the expression must yield a value of at least 256.

Number-of-files is an *address-expression* which gives the maximum number of files that may be handled at once. If omitted the current setting is retained.

Maximum-record-length is an *address-expression* which specifies the maximum random record size. If omitted the current setting is retained.

Notes.

The syntax means that it is only necessary to give values for those settings which are to be changed.

Changing the stack size causes BASIC to forget any active FOR, WHILE or GOSUB commands.

Reducing the maximum number of files automatically closes any files associated with file numbers greater than the new maximum.

Changing the maximum random record size automatically closes all active files. Keyed files closed this way are not marked consistent, so should be explicitly CLOSED beforehand.

BASIC's stack is used during expression evaluation and to hold information for FOR and WHILE loops and for GOSUB returns. The default size of 512 bytes should be adequate for most programs, unless extremely deeply nested FOR or WHILE loops or GOSUBs are used.

Associated Keywords.

CLEAR, HIMEM, OPEN, FRE

MERGE

Merge program from disc into current program. **Command**
Unavailable in Run Only versions

Use.

To add the contents of a file to the current program in memory.

Form.

MERGE *string-expression*

The *string-expression* yields the name of the file to be read. If no type extension is given .BAS is assumed.

Notes.

MERGE has the following effects on the current environment:

- Discards all variables
- All User Functions are forgotten.
- ON ERROR GOTO is turned off.
- All open files are retained.
- RESTORE action is taken.
- OPTION BASE, DEFINT, DEFSNG, DEFDBL and DEFSTR settings are reset.

During a MERGE operation a line in the new program with the same number as an existing line will replace the existing line.

BASIC returns to Direct Mode after executing a MERGE command.

If a protected program is MERGED with an unprotected program, the result is a protected program.

Associated Keywords.

LOAD, CHAIN MERGE, SAVE, CHAIN, RUN

**Replace part of a string.
Return a part of a string.**

**Command
Function**

Use.

MID\$ specifies part of a string (a sub-string) which can be used either as the destination of an assignment (MID\$ as a Command) or as an argument in a string expression (MID\$ as a Function).

Form.

MID\$ (*string*, *start-position*[, *sub-string-length*])

For MID\$ as a Command *string* must be a *string-variable*, part of which is to be altered.

For MID\$ as a Function *string* is a *string-expression*, part of which will be returned as the function's value.

The *start-position* is an *integer-expression* which specifies the character in *string* which is to be the first character of the sub-string. The *integer-expression* must yield a value in the range 1 to 255.

The *sub-string-length* is an *integer-expression* which specifies the length of the sub-string. If omitted, or greater than the number of characters after the *start-position*, the sub-string extends to the end of the original string. The *integer-expression* must yield a value in the range 0...255.

Notes.

The sub-string specified in MID\$ is defined by a starting character position and a length. The first character of the original string is at position 1. The length defaults to all characters after the starting position of the sub-string. If the starting position specified is beyond the end of the string, the sub-string is null. The sub-string cannot extend beyond the end of the original string, so the effective length of the sub-string may be less than the specified length.

When MID\$ is used as a Command it appears on the lefthand side of an assignment. On the righthand side must be a *string-expression*. The sub-string specified by the MID\$ is replaced by the value of the *string-expression*. If the *string-expression* yields a string that is shorter than the sub-string, then the extra characters in the sub-string are unaffected. If the *string-expression* yields a string that is longer than the sub-string, then the excess characters are discarded.

Associated Keyword.

LSET, RSET, LEFT\$, RIGHT\$, FIELD

MIN

Determine minimum value.

Function

Use.

To determine the minimum of a number of values.

Form.

MIN (*list-of: numeric-expression*)

Notes.

MIN returns the value of the smallest of the *numeric-expression(s)*.

Associated Keywords.

MAX

Make Double Length String.

Function

Use.

To convert a Double Length value into a eight byte string, so that the value may be stored in that form. This is particularly useful for storing Double Length values in random records.

Form.

MKD\$ (*numeric-expression*)

Notes.

If the expression does not yield a Double Length result, then it is converted to Double Length.

It is not recommended that the string be processed in any way, other than to assign it to another string. The function CVD converts the string back to a Double Length value.

This function does not perform any binary to decimal conversion, so the string is an exact representation of the Double Length value.

Associated Keywords.

CVD, MKI\$, MKS\$

MKDIR

Make a new directory.

Command

Use.

For Mallard-86 MSDOS 2 version, this makes a new directory on the specified, or default drive.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all ignore this command.

Form.

MKDIR *string-expression*

where the *string-expression* yields a *dir-name*, the last *dir-simple* of which specifies the directory to make.

Notes.

The directory must not already exist.

Associated Keywords.

CHDIR, RMDIR, CD, MD, RD, CHDIR\$, FINDDIR\$

Make Integer String.

Function

Use.

To convert an Integer value into a two byte string, so that the value may be stored in that form. This is particularly useful for storing Integer values in random records.

Form.

MKI\$ (*numeric-expression*)

Notes.

If the expression does not yield an Integer result, then it is rounded to Integer.

It is not recommended that the string be processed in any way, other than to assign it to another string. The function CVI converts the string back to an Integer value.

This function does not perform any binary to decimal conversion.

MKIK\$ and MKUK\$ are similar functions which produce two byte strings suitable for use as, or in, Jetsam keys. The string produced by MKI\$ does not sort correctly.

Associated Keywords.

CVI, MKS\$, MKD\$

Make Integer Key String.

Function

Use.

To convert an Integer value into a two byte string suitable for use as a Jetsam key, or as part of one.

Form.

MKIK\$ (*numeric-expression*)

Notes.

If the expression does not yield an Integer result, then it is rounded to Integer.

It is not recommended that the string be processed in any way, other than to assign it to another string. The function CVIK converts the string back to an Integer value.

This function does not perform any binary to decimal conversion.

MKIK\$ should be used to create strings to be used as Jetsam keys for integer values in the range -32768...32767. There are advantages to keeping Jetsam keys as short as possible, and this function produces a compact representation for the key. The function MKUK\$ should be used for integer values in the range 0...65535.

The function MKI\$ produces a similar sort of string, but not one which will sort correctly if used as, or in, a Jetsam key.

Associated Keywords.

CVIK, MKI\$, MKUK\$

Make Single Length String.

Function

Use.

To convert a Single Length value into a four byte string, so that the value may be stored in that form. This is particularly useful for storing Single Length values in random records.

Form.

MKS\$ (*numeric-expression*)

Notes.

If the expression does not yield a Single Length result, then it is converted to Single Length.

It is not recommended that the string be processed in any way, other than to assign it to another string. The function CVS converts the string back to a Single Length value.

This function does not perform any binary to decimal conversion, so the string is an exact representation of the Single Length value.

Associated Keywords.

CVS, MKI\$, MKD\$

Make Unsigned Integer Key String.

Function

Use.

To convert an Unsigned Integer value into a two byte string suitable for use as a Jetsam key, or as part of one.

Form.

MKUK\$ (*numeric-expression*)

Notes.

If the expression does not yield an Integer result, then it is rounded to Integer.

It is not recommended that the string be processed in any way, other than to assign it to another string. The function CVUK converts the string back to a numeric value.

This function does not perform any binary to decimal conversion.

MKUK\$ should be used to create strings to be used as Jetsam keys for integer values in the range 0...65535. There are advantages to keeping Jetsam keys as short as possible, and this function produces a compact representation for the key. The function MKIK\$ should be used for integer values in the range -32768...32767.

The function MKI\$ produces a similar sort of string, but not one which will sort correctly if used as, or in, a Jetsam key.

Associated Keywords.

CVUK, MKI\$, MKIK\$

Rename disc file.

Command

Form.

NAME *file-name-expression* AS *file-name-expression*

The first *file-name-expression* gives the name of the file whose name is to be changed.

The second *file-name-expression* gives the new name for the file.

Notes.

The file must exist. No file with the new name may exist.

Renaming an open file is not recommended. On multiuser systems this command may fail if the file is in use by other users or if the file is locked.

Associated Keywords.

REN

NEW

Prepare to enter new program.

Command

Unavailable in Run Only versions

Use.

Before starting a new program NEW may be used to completely clear the current contents of memory.

Form.

NEW

Notes.

As well as deleting the program in memory, the following are affected:

- All variables are discarded.
- All User Functions are forgotten.
- ON ERROR GOTO is turned off.
- All files are closed.
- OPTION BASE, DEFINT, DEFSNG, DEFDBL and DEFSTR settings are reset.
- TRON is turned off.

BASIC always returns to Direct Mode after a NEW is executed.

NEXT

Step FOR control variable to next value.

Command

Use.

A NEXT marks the end of a FOR loop. The NEXT command may be anonymous, or may refer to its matching FOR.

Form.

NEXT [*list-of: variable*]

where: NEXT *variable, list-of: variable*

is equivalent to: NEXT *variable: NEXT list-of: variable*

Notes.

A NEXT command defines the end of a FOR loop. The way in which FOR and NEXT are tied together is described under FOR. When a NEXT command is encountered BASIC knows which (if any) FOR it must be associated with. If the NEXT is followed by a variable name, then that name must be the same as the name of the control variable in the matching FOR command.

It is an error to execute a NEXT which is not associated with an active FOR.

Associated Keywords.

FOR

Octal string.

Function

Use.

To produce a string of octal digits representing the value of the given expression.

Form.

OCT\$ (*unsigned-integer-expression*[, *field-size*])

The value of the *unsigned-integer-expression* is treated as an unsigned 16 bit integer, to be converted into a string of octal characters. See Section 2.10 for a description of unsigned integer.

The optional *field-size* is an *integer-expression* which gives the minimum size of string to be produced, and must yield a value in the range 0...16.

Notes.

OCT\$ always generates as many characters as are required to represent the number (zero generates at least one digit). If the optional *field-size* parameter is present and the string is too short, then the string is filled to length with leading zeros. The string is not truncated if it is too long.

Associated Keywords.

HEX\$, DEC\$, STR\$

ON expression GOSUB/GOTO

Computed GOSUB & GOTO.

Command

Use.

To choose one of a number of subroutines to call or a number of lines to jump to, depending on the result of an expression.

Form.

ON *integer-expression* GOSUB *list-of: line-number*

ON *integer-expression* GOTO *list-of: line-number*

The *integer-expression* must yield a value in the range 0...255.

Notes.

The result of the *integer-expression* selects a line number from the list: 1 selects the 1st, 2 the 2nd, and so on. The subroutine starting at the selected line number is called, or the selected line number is jumped to. Zero, or any result greater than the number of line numbers in the list, will do nothing.

Null line number entries in the list are legal. Selecting a null entry is not (Error 2).

Associated Keywords.

GOTO, GOSUB

ON ERROR GOTO

Set Error Trap.

Command

Use.

When BASIC detects an error while obeying a command it can either take the default action, which is to generate an error message and return to Direct Mode, or it can invoke an error handling subroutine in the program. ON ERROR GOTO sets BASIC into one of these states.

Forms.

ON ERROR GOTO *line-number*

ON ERROR GOTO 0

The *line-number* specifies the line to which control is to be transferred in the event of an error.

Notes.

The form ON ERROR GOTO *line-number* enables error trapping. ON ERROR GOTO 0 disables error trapping – note the special effect of this during error processing, see below.

If an error occurs in Program Mode while error trapping is enabled control is transferred to the line number specified in the ON ERROR GOTO command. BASIC is now in Error Processing Mode. The variables ERR and ERL are set to indicate which error has occurred, and which line it occurred in.

ON ERROR GOTO 0 in Error Processing Mode not only disables error trapping, but also causes BASIC to take the default error action for the error that was trapped. Error processing routines may, therefore, cope with those errors for which they are competent and invoke the default action for the rest.

If an error is detected during Error Processing Mode BASIC immediately takes the default action.

The RESUME command is only legal during Error Processing Mode, and allows the program to resume execution in one of three ways:

- at the statement in which the error occurred
- at the statement after the one in which the error occurred
- at a specified line number

Associated Keywords.

ERR, ERL, RESUME

OPEN

Open file.

Command Extended in Jetsam Versions

Use.

For random and sequential files, to open a disc file and associate it with a file number, so that the file may be accessed.

For keyed files, to open both the index and data files for keyed index access and associate them with a file number for keyed access.

Form – for Random and Sequential Files.

OPEN *mode*, *file-reference*, *filename-expression* [, *record-length*]
or OPEN *mode*, *file-reference*, *filename-expression*, [*record-length*], *lock*

Notes.

Mode is a *string-expression* specifying the mode of access to the file. The first character of the resulting string must be one of:

- I – for Sequential Input
- O – for Sequential Output
- R – for Random access

The *file-reference* gives the file number with which the file is to be associated, and by which it will be referenced. It is an error (Error 55) to attempt to use a file number which is currently in use.

The *filename-expression* gives the name of the file to be opened.

A *record-length* may be specified if the *mode* is "R". *record-length* is an *integer-expression*, which must yield a value in the range 1...*max*. *max* is set by CLEAR and MEMORY command, and when BASIC is first loaded (default value 128). If no *record-length* is specified, 128 is assumed.

If a *lock* parameter is present it specifies the file lock to be applied when the file is open. On single user systems this parameter has no effect. On multiuser systems this parameter over-rides the default file lock.

Note – for Random and Sequential Files.

When opening a file for input the file must exist.

When opening a file for output any existing file of the same name is immediately erased. A new file of the required name is created.

When opening a file for random access the file is created if it does not already exist.

On multiuser systems if no lock parameter is given BASIC assumes a default lock, depending on the open *mode*. If the *lock* parameter is present the type of lock is restricted, also depending on the open *mode*. Thus:

- Input files may only be read or write locked, the default is read locked. (Write locking gives exclusive access to the file; it may not be written to.)
- Output files may only be write locked and the default is write locked.
- Random files may have any lock applied, the default is write locked.

Form – for Keyed Files.

```
OPEN "K", file-reference, data-filename, index-filename,  
lock[, record-length-variable]  
or OPEN "K", file-reference, data-filename, index-filename,  
lock[, record-length-variable], user-string-variable
```

The *data-filename* is a *filename-expression* giving the name of the data file to be opened.

The *lock* specified is applied to the keyed file when it is opened.

The *user-string-variable*, if present, is a *string-variable* which is set to the user string specified when the file was created.

The open may fail in a number of ways:

- error 53: if either file does not exist.
- error 73: if the required file lock cannot be obtained.
- error 113: if either file was not created using a CREATE command (ie. the file is not recognised by Jetsam).
- error 115: the files are not consistent (ie. the keyed file was not closed properly the last time it was modified).

Associated Keywords.

314

OPTION BASE

Set the base value for array subscripts.

Command

Use.

Array subscripts may start from zero or one. The OPTION BASE command chooses between these alternatives.

Form.

OPTION BASE *integer-expression*

where the *integer-expression* must yield a value in the range 0...1.

Notes.

Attempting to obey more than one OPTION BASE command will generate an error (Error 10).

Attempting to obey an OPTION BASE command while any array exists will generate an error (Error 10).

The default base value is 0.

Associated Keywords.

DIM, ERASE, CLEAR

OPTION FIELD

Make reserved bytes part of user record. Jetsam Command

Use.

To make the two bytes of a keyed record which are reserved by Jetsam into part of the user record. The bytes must then be defined in a FIELD command.

Form.

OPTION FIELD *numeric-expression*

Notes.

If *numeric-expression* = 2, the two bytes of the keyed record are hidden from the user. This is the default case.

If *numeric-expression* = 0, FIELD statements must explicitly allow for two bytes reserved for Jetsam at the start of each record. This is compatible with older versions of BASIC.

Warning.

This command is provided solely for compatibility with older versions of BASIC and should not normally be used.

Associated Keywords.

FIELD

OPTION FILES

Set the default drive and user.

Command

Use.

To change the default drive and the default user number.

Form.

OPTION FILES *string-expression*

where *string-expression* is a valid user number or drive letter. The user number or drive letter may optionally be omitted.

Notes.

This command is only valid in CP/M like operating systems.

The default drive and user are changed until changed again by BASIC or control is returned to the operating system. On return to the operating system, the default user and drive are set to their values when BASIC was invoked.

OPTION INPUT

Set trap for all console input.

Command

Use.

To nominate a machine language subroutine to be called to input characters from the console. The nominated subroutine is used instead of the usual call to the operating system for all console input.

Form.

OPTION INPUT = *address-expression, address-expression*

or OPTION INPUT

The first *address-expression* gives the address of the nominated machine language 'Test Keyboard Status' subroutine.

The second *address-expression* gives the address of the nominated machine language 'Fetch Keyboard Character' subroutine.

Notes.

The first form sets new subroutines to be called. The second form and the RUN command cancel any previously nominated subroutines and BASIC reverts to using the usual operating system calls.

The two subroutines are:

- 'Test Keyboard Status'

This routine is called on a regular basis by BASIC to check if anything has been typed, so that Control-C and Control-S can be detected. This check is an overhead which affects the speed of BASIC, so this routine should be made as fast as possible.

- 'Fetch Keyboard Character'

This routine is called when BASIC requires a character from the keyboard. The routine must wait until a character is available, and return it.

The interface requirements for these routines are very strict. Any routine which deviates from them is likely to cause BASIC to fail. The routines may use a modest amount of stack.

On Mallard-80, the interfaces for the subroutines are:

'Test Keyboard Status'

Entry: No conditions

Exit: If a character is available:
Carry Flag True
A = Character from keyboard

If no character is available:
Carry Flag False
A may be corrupted

Always:
BC, DE, HL and other Flags may be corrupted
All other registers must be preserved

'Fetch Keyboard Character'

Entry: No conditions

Exit: A = Character from keyboard
BC, DE, HL and Flags may be corrupted
All other registers must be preserved

On Mallard-86:

The *address-expressions* give the segment offsets of the address of the subroutines. The segment part of the addresses is set to the DEF SEG setting at the time of the OPTION INPUT command. The interfaces for the subroutines are:

'Test Keyboard Status'

Entry: CS contains the segment part of the subroutine address
DS, ES & SS all contain the base of BASIC's data segment.

Exit: If a character is available:
Carry Flag True
AL = Character from keyboard

If no character is available:
Carry Flag False
AL may be corrupted

Always:
AH and other Flags may be corrupted
All other registers must be preserved

'Fetch Keyboard character'

- Entry: CS contains the segment part of the subroutine address
DS, ES & SS all contain the base of BASIC's data segment.
- Exit: AL = Character from keyboard
AH and Flags may be corrupted
All other registers must be preserved

The subroutines are called using FAR CALL, so must return using a FAR RETURN.

Associated Keywords.

OPTION LPRINT, OPTION PRINT, DEF SEG

On Mallard-80, the interfaces for the subroutines are:

'Test Keyboard Status'

Entry: No conditions

Exit: If a character is available:
Carry Flag True
A = Character from keyboard

If no character is available:
Carry Flag False
A may be corrupted

Always:
BC, DE, HL and other Flags may be corrupted
All other registers must be preserved

'Fetch Keyboard Character'

Entry: No conditions

Exit: A = Character from keyboard
BC, DE, HL and Flags may be corrupted
All other registers must be preserved

On Mallard-86:

The *address-expressions* give the segment offsets of the address of the subroutines. The segment part of the addresses is set to the DEF SEG setting at the time of the OPTION INPUT command. The interfaces for the subroutines are:

'Test Keyboard Status'

Entry: CS contains the segment part of the subroutine address
DS, ES & SS all contain the base of BASIC's data segment.

Exit: If a character is available:
Carry Flag True
AL = Character from keyboard

If no character is available:
Carry Flag False
AL may be corrupted

Always:
AH and other Flags may be corrupted
All other registers must be preserved

'Fetch Keyboard character'

- Entry:** CS contains the segment part of the subroutine address
DS, ES & SS all contain the base of BASIC's data segment.
- Exit:** AL = Character from keyboard
AH and Flags may be corrupted
All other registers must be preserved

The subroutines are called using FAR CALL, so must return using a FAR RETURN.

Associated Keywords.

OPTION LPRINT, OPTION PRINT, DEF SEG

OPTION LPRINT

Set trap for all printer output.

Command

Use.

To nominate a machine language subroutine to be called to output characters to the printer. The nominated subroutine is used instead of the usual call to the operating system for all output to the printer.

Form.

OPTION LPRINT = *address-expression*

or OPTION LPRINT

The *address-expression* gives the address of the nominated machine language subroutine.

Notes.

The first form sets a new subroutine to be called. The second form and the RUN command cancel any previously nominated subroutine and BASIC reverts to using the operating system call.

The interface requirements for this routine are very strict. Any routine which deviates from them is likely to cause BASIC to fail. The routine may use a modest amount of stack.

On Mallard-80 the interface for the subroutine is:

Entry: C contains the character to be sent to the printer

Exit: A, BC, DE, HL registers and all Flags may be corrupted

On Mallard-86:

The *address-expression* gives the segment offset of the address of the subroutine. The segment part of the address is set to the DEF SEG setting at the time of the OPTION LPRINT command. The interface for the subroutine is:

Entry: CL contains the character to be sent to the printer

CS contains the segment part of the subroutine address

DS, ES & SS all contain the base of BASIC's data segment

Exit: AX and Flags may be corrupted

All other registers must be preserved

The subroutine is called using a FAR CALL, so must return a FAR RETURN.

Associated Keywords.

OPTION INPUT, OPTION PRINT, LPRNT, DEF SEG

OPTION NOT TAB

Turn off Tab expansion.

Command

Use.

To prevent BASIC from transforming Tab (&H09) characters into strings of spaces when printing or displaying them.

Form.

OPTION NOT TAB

Notes.

BASIC usually transforms Tab (&H09) characters sent to the console or the printer into enough spaces to move forward to the next tab position – where tabs are set at every eighth column. OPTION NOT TAB overrides this action, so that Tab characters are sent to the printer or the console without change. It can therefore be used to prevent escape sequences that include &H09 from being transformed into an incorrect series of values.

OPTION NOT TAB does not affect the TAB Print Function, or BASIC's Print Zone operations.

OPTION NOT TAB takes effect until an OPTION TAB or a RUN command is executed.

Associated Keywords.

PRINT, LPRINT, WRITE, OPTION TAB, RUN

OPTION PRINT

Set trap for all console output.

Command

Use.

To nominate a machine language subroutine to be called to output characters to the console. The nominated subroutine is used instead of the usual call to the operating system for all output to the console.

Form.

OPTION PRINT = *address-expression*

or OPTION PRINT

The *address-expression* gives the address of the nominated machine language subroutine.

Notes.

The first form sets a new subroutine to be called. The second form and the RUN command cancel any previously nominated subroutine and BASIC reverts to using the usual operating system call.

The interface requirements for this routine are very strict. Any routine which deviates from them is likely to cause BASIC to fail. The routine may use a modest amount of stack.

On Mallard-80 the interface for the subroutine is:

Entry: C contains the character to be sent to the console

Exit: A, BC, DE & HL registers and Flags may be corrupted

On Mallard-86:

The *address-expression* gives the segment offset of the address of the subroutine. The segment part of the address is set to the DEF SEG setting at the time of the OPTION LPRINT command. The interface for the subroutine is:

Entry: CL contains the character to be sent to the console

CS contains the segment part of the subroutine address

DS, ES & SS all contain the base of BASIC's data segment

Exit: AX and Flags may be corrupted

All other registers must be preserved

The subroutine is called using a FAR CALL, so must return a FAR RETURN.

Associated Keywords.

OPTION INPUT, OPTION LPRINT, PRINT, DEF SEG

OPTION RUN

Prevent BASIC program from being suspended or stopped Command

Use.

OPTION RUN disables the effect of Control-C and Control-S from the keyboard.

Form.

OPTION RUN

Notes.

Control-C usually stops the current program and returns to Direct Mode. Control-S usually suspends the current program until another key is pressed. After an OPTION RUN command neither key has any effect. Furthermore, since BASIC no longer has to test for any key the program will run faster – usually BASIC checks the keyboard after each statement, which imposes a system dependent overhead. (In Run-Only versions neither Control-C nor Control-S ever have any effect).

OPTION RUN interacts with INKEY\$. Usually INKEY\$ cannot return Control-C or Control-S, since BASIC reacts to Control-C and Control-S typed at INKEY\$ – if it did not, the effect of these keys would be unpredictable in a program using INKEY\$. After OPTION RUN, INKEY\$ will return both Control-C and Control-S, since there can no longer be any confusion.

Run-Only versions of BASIC ignore Control-C and Control-S for compatibility reasons. So the only effect of OPTION RUN or OPTION STOP is to alter the action of INKEY\$ when presented with Control-C or Control-S. In OPTION STOP mode, these characters are never returned; in OPTION RUN mode, they will be returned if typed.

The current OPTION RUN/STOP setting does not affect the STOP, END or SYSTEM commands or the way in which errors are handled.

OPTION STOP

Reverses the effect of OPTION RUN.

Command

Use.

OPTION STOP will re-enable the effects of Control-C and Control-S if they have been disabled by OPTION RUN.

Form.

OPTION STOP

Notes.

In OPTION STOP mode, INKEY\$ will never return the characters Control-C and Control-S.

The current OPTION RUN/STOP setting does not affect the STOP, END or SYSTEM commands or the way in which errors are handled.

Associated Keywords.

OPTION RUN

OPTION TAB

Turn on Tab expansion.

Command

Use.

To re-enable BASIC's transformation of Tab (&H09) characters into strings of spaces when printing or displaying them.

Form.

OPTION TAB

Notes.

BASIC usually transforms Tab (&H09) characters into enough spaces to move forward to the next tab position – where tabs are set at every eighth column. The OPTION NOT TAB overrides this action, so that Tab characters are sent to the printer or the console without change.

OPTION TAB re-enables the transformation of Tab characters if this has been disabled by an OPTION NOT TAB since the last RUN command was executed.

Associated Keywords.

PRINT, LPRINT, WRITE, OPTION NOT TAB, RUN

OSERR

Return operating system dependent error indication.

Function

Use.

To further identify the cause of a BASIC error 21.

Form.

OSERR

Notes.

Some operating systems return a number of miscellaneous but interesting errors to BASIC. Rather than have a different set of BASIC errors for each such operating system BASIC provides the single error number 21 and the function OSERR. When an error 21 has occurred the function OSERR will return an operating system dependent number which will identify the error.

OSERR returns a value in the range 0...65535.

Associated Keywords.

ERR, ERL, ON ERROR GOTO

OUT OUTW

Output a value to a processor output port. **Command**

Use.

To send a value to a given I/O port.

Form.

OUT *port-number, integer-expression*

OUTW *port-number, integer-expression*

For Mallard-80 the *port-number* is an *integer-expression* which must yield a value in the range 0...255.

For Mallard-86 the *port-number* is an *address-expression*.

The *integer-expression* in OUT gives the data byte to be sent to the given output port. The expression must yield a value in the range 0...255.

The *address-expression* in OUTW gives the data word to be sent to the given word output port.

Notes.

OUTW is illegal in Mallard-80.

Associated Keywords.

INP, INPW, WAIT, WAITW

PEEK

Peek at memory location.

Function

Use.

To read a given byte of the machine's memory.

Form.

`PEEK (address-expression)`

Notes.

PEEK returns a value in the range 0...255.

In Mallard-86 the *address-expression* gives the offset part of the address to read from. The segment part is given by the User Segment Base most recently defined by DEF SEG.

Associated Keywords.

POKE, DEF SEG

POKE

Poke a value into machine memory.

Command

Use.

Allows direct write access to machine memory.

Form.

POKE *address-expression*, *integer-expression*

The first *address-expression* gives the address of the byte to be written to.

The *integer-expression* gives the value to be written to the byte at the given address. The expression must yield a value in the range 0...255.

Notes.

In Mallard-86 the *address-expression* gives the offset part of the address to write to. The segment part is given by the User Segment Base most recently defined by DEF SEG.

Associated Keywords.

PEEK

Console position.

Function

Use.

To establish the current print position on the console.

Form.

POS (*numeric-expression*)

where *numeric-expression* is a dummy argument, so POS(0) is the recommended form.

Notes.

POS returns a logical position on the console, which may bear no relation whatsoever to the current position of the cursor!

In calculating the logical position all non-printing characters (those with values less than &H20) are not counted, except for:

- Backspace (&H08), which counts – 1 unless the position is one.
- Tab (&H09), when expanded to spaces counts one for each space. (Tab positions 8 characters apart all the way across the console are implicitly defined).
- Carriage return (&H0D), which sets the position to one.

The position is also affected by new lines inserted by BASIC if the logical position exceeds the width of the console, as set by WIDTH.

If the width is set infinite (by WIDTH 255) then the logical position will grow to 255, but no further.

Associated Keywords.

LPOS, WIDTH

PRINT

Print to console.

Command

Use.

To print data, numeric or string, to the console.

Form.

PRINT [*print-list*][*using-clause*][*separator*]

print-list is: *print-item*[*separator print-item*]*

where *print-item* is: *expression*
or: SPC (*integer-expression*)
or: TAB (*integer-expression*)

using-clause is: USING *string-expression*; *using-list*

where *using-list* is: *expression*[*separator expression*]*

separator is: comma or semicolon.

(Note that the [...]* metalanguage construct means that the object is optional, but may be repeated any number of times.)

Notes.

See Chapter 6 for a description of print facilities.

The *print-item*(s), if any, in the *print-list* are evaluated and printed in Free Format. A comma following a *print-item* causes BASIC to advance to the beginning of the next Print Zone after the item is printed. A semicolon serves only to separate the items, and has no effect on the output.

If a *using-clause* is present, then the *string-expression* defines a template which controls the format in which the values of the expressions in the *using-list* are printed. Commas or semicolons may be used to separate the items in the *using-list*, neither has any effect on the output.

When all the arguments have been processed a new line is started, unless the PRINT command ends in a *separator* or TAB or SPC.

Associated Keywords.

LPRINT, POS, PRINT #, WIDTH, ZONE

PRINT

Print to file.

Command

Use.

To put data to a file, formatted in the same way as PRINTing to the console.

Form.

`PRINT #filename-expression[, print-arguments]`

The *filename-expression* specifies the file to be printed to. The file must be open for Output or for Random access.

The *print-arguments* take precisely the same form as in the PRINT command. If no arguments are given, a Carriage Return, Line Feed pair is written.

Notes.

See Chapter 6 for a description of print facilities.

PRINT # is the same as PRINT, except that the output is sent to a file, not to the console, and that lines in a file are considered to be infinitely wide. The handling of the *print-arguments* is identical, except that Tab characters (value &H09) are not expanded, and do not affect the logical position.

It is possible to use PRINT # to output to a random access file. Characters are put to the random record buffer. BASIC maintains an internal pointer associated with the buffer, which is used to keep track of where to write next. This pointer is set to the start of the buffer when the file is opened and subsequently by GET and PUT commands. It is an error to attempt to overfill the random record buffer.

PRINT # does not work with keyed files.

Associated Keywords.

PRINT, WRITE #

PUT

Put data to random access file.

Command

Use.

To put the contents of the random record buffer to the given record in the given random or keyed data file.

Form.

`PUT file-reference[, record-number]`

or `PUT file-reference, [record-number], lock`

The *file-reference* specifies the file to be written to, which must be an open random or keyed file.

For random files: The *record-number* specifies the record to be written. If no number is specified the record following the one in the last GET or PUT is assumed – if this is the first GET or PUT then record 1 is assumed.

The *lock* parameter, if present, specifies the record lock to be applied to the record after it is written. This parameter may only be present when writing to random files.

For keyed files: The *record-number* specifies which record in the data file is to be written. If the parameter is omitted the record number given by the current position is used.

For Random Files Note.

An internal pointer is associated with the random record buffer, to allow PRINT # et al to put data to the record. A PUT operation resets that pointer.

For Keyed Files Note.

The record, or the file, must be write locked.

Only record numbers that have been returned by the FETCHREC function should be used.

When a record is PUT, a check is made to ensure that the information in the first two bytes of the record is valid. The operation of ADDKEY and DELKEY alter this information, so it is important that no ADDKEY or DELKEY is performed between a GET and a PUT.

For Random Files in Multiuser Versions Note.

PUT releases any outstanding temporary record write lock. See Section 9.3.4 for a description of the temporary record write lock mechanism.

Associated Keywords.

PRINT #, WRITE #, GET

RANDOMIZE

Randomise the current random number seed.

Command

Use.

BASIC's random number generator produces a pseudo-random sequence, in which each number depends completely on the previous number. Starting from a given value the sequence is always the same. RANDOMIZE sets a new initial value for the random number generator, either to a given value, or to a value entered by the operator.

Form.

RANDOMIZE [*integer-expression*]

If the expression is omitted then the operator is prompted:

Random Number Seed ?

to get a suitable value, which may be any form of number.

Notes.

Setting the initial random number to the same number will result in the same sequence being generated.

Associated Keywords.

RND

Specify whether rank may have duplicate keys. **Function**

Use.

To set whether a rank may have keys added which have identical values to existing keys.

Form.

RANKSPEC (*file-reference*, *rank*, *unique*)

where:

file-reference gives the file number of an open keyed file.

rank gives the rank number being affected.

unique is a *numeric-expression* which has the value 0 when duplicate keys may be added for the given *rank*. If it has the value 1 then duplicate keys may not be added.

Return Codes.

0	Success	the <i>rank</i> has a new <i>unique</i> property
130	Failure	the file is read locked by another user

Notes.

The RANKSPEC setting is stored in the file header. It remains set until another RANKSPEC is issued. Newly created files are automatically set up to allow the use of duplicate keys.

If duplicates are not allowed then ADDKEY and ADDREC fail with the Return Code 116. The duplicate is not added and the current position is set to the duplicate within the file.

Setting RANKSPEC does not check whether a file currently contains duplicate keys – it merely affects their addition in the future.

The RANKSPEC function causes the keyed file to be marked inconsistent. It will be marked consistent again if it is successfully closed or CONSOLIDATED.

Associated Keywords.

ADDKEY, ADDREC

Remove a directory.

Command

Use.

For Mallard-86 MSDOS 2 version, this removes a directory from the specified, or default drive.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all ignore this command.

Form.

RD *dir-name*

where the *dir-name* specifies the directory to remove.

Notes.

The directory must exist and be empty, except for . and ..

The directory must not be the current directory.

This command takes the rest of the current line as its argument, irrespective of colons or single quotes.

Associated Keywords.

CHDIR, MKDIR, RMDIR, CD, MD, CHDIR\$, FINDDIR\$

READ

Read from DATA statements.

Command

Use.

READ fetches data from DATA statements and assigns it to variables.

Form.

READ *list-of: variable*

Notes.

All the DATA statements in a program, taken in order, form a list of constants. These constants may be read into variables by the READ command. READ steps on to the next constant in the list and assigns it to the next variable in the list of variables given. The constant and the variable to which it is read must be compatible. Attempting to READ past the last DATA constant generates an error (Error 4).

Associated Keywords.

DATA, RESTORE

REM

Remark.

Command

Use.

To put comments into BASIC programs.

Form.

REM rest-of-line

BASIC ignores the rest of the line after the REM. Note that colon (statement separator) is also ignored.

Notes.

REM 'commands' may be executed, and have no effect – execution continues with the first command in the next line. It is permissible to transfer control to a REM command, by GOTO or by GOSUB.

A single quote character in a line (not in a string) is equivalent to :REM, and may also prefix comment text, except in:

- DATA commands where a single quote is treated as part of an unquoted string.
- DEL, ERA, DIR, REN and TYPE commands, where the rest of the line is taken as the command's parameter.

Some languages have a remainder operator called REM. This is a fruitful source of syntax errors in BASIC programs. BASIC has an operator called MOD.

REN

Rename file.

Command

Use.

To change the name of a disc file.

Form.

– in versions for CP/M-type operating systems:

REN *new-name=old-name*

– in versions for MS-DOS (PC-DOS) type operating systems:

REN *old-name new name*

The *old-name* is a *file-name* specifying the file the name of which is to be changed.

The *new-name* is a *file-name* giving the new name for the file.

Notes.

The file must exist. No file with the new name may exist.

REN takes the rest of the current line as its arguments, irrespective of colons or single quotes.

Renaming an open file is not recommended. On multiuser systems this command may fail if the file is in use by other users or if the file is locked.

Associated Keywords.

NAME

RENUM

Renumber the current program.

Command

Unavailable in Run-Only versions

Use.

To renumber all or part of a program.

Form.

RENUM

or RENUM *new-line-number*

or RENUM [*new-line-number*], *old-line-number*

or RENUM [*new-line-number*], [*old line-number*], *increment*

If no parameters are given this is equivalent to RENUM 10, , 10.

new-line-number is a *line-number*, giving the new number for the first line to be renumbered. If omitted 10 is assumed.

old-line-number is a *line-number*, giving the first line to be renumbered. If omitted the first line of the program is assumed.

increment is a *line-number*, giving the increment to use between the new line numbers. If omitted 10 is assumed.

Notes.

RENUM adjusts references to line numbers throughout the program to refer to the changed line numbers. Line number references in the following commands are affected:

ELSE, GOSUB, GOTO, LIST, LLIST, RESTORE, RESUME, RUN, THEN, DELETE and the DELETE in CHAIN MERGE

(see also ERL which interacts with RENUM).

When part of the program is renumbered RENUM disallows any attempt to change the order of lines, or to use line numbers already in use in the part of the program which is not being renumbered.

If RENUM discovers line number references to lines that do not exist BASIC produces the message:

Undefined line 99999 in 88888

where: 99999 is the line number which is referred to, but does not exist. The reference is left unchanged.

and: 88888 is the new number of the line in which the reference occurs.

After a RENUM command BASIC returns to Direct Mode.

RESET

Reset file system.

Command

Use.

To reset the file system so that discs may be changed.

Form.

RESET [*string-expression*]

Notes.

The optional *string-expression* indicates which drives are to be reset – all the files on these drives are closed. The *string-expression* specifies the drive: its value must be a valid drive letter for the operating system that is being used. The *string-expression* may not be null.

RESET without a parameter resets all drives and closes all files.

Any open keyed files are not marked consistent, so these should be closed explicitly beforehand.

On multiuser systems this command may fail as other users may be using files, preventing a reset of the disc filing system.

Associated Keywords.

CLOSE

RESTORE

Restore pointer into DATA list.

Command

Use.

To move the internal pointer to the next DATA item to a given position (typically back to the first DATA statement).

Form.

RESTORE [*line-number*]

The *line-number* gives the line to which the DATA pointer is to be set. If omitted the pointer is set back to the start of the program.

Notes.

READ commands move a pointer to DATA items through the program. The RESTORE command moves that pointer to a specified position in the program. The next READ command executed will search forward from that point for the next DATA statement, and start reading from there.

Associated Keywords.

READ, DATA

RESUME

Resume execution after processing an error. **Command**
Legal only in Error Processing Mode

Use.

When an error has been trapped by an ON ERROR GOTO routine, and has been processed by it, RESUME allows normal execution to continue, from a variety of points.

Form.

RESUME
or RESUME *line-number*
or RESUME NEXT

Notes.

RESUME is legal only during Error Processing Mode (ie. in an ON ERROR GOTO routine).

RESUME with no parameter returns control to the beginning of the statement in which the error was detected.

RESUME *line-number* returns control at the specified line.

RESUME NEXT returns control to the statement immediately after the statement in which the error was detected.

Associated Keywords.

ON ERROR GOTO

RETURN

Return from subroutine.

Command

Use.

Signals the end of a subroutine, BASIC returns to continue processing after the GOSUB which invoked it.

Form.

RETURN

Notes.

It is permissible to RETURN from a subroutine during a FOR or WHILE loop. RETURNing in this way abandons any active FOR or WHILE loops started since the GOSUB which invoked the subroutine.

Associated Keywords.

GOSUB, ON x GOSUB

RIGHT\$

Extract righthand part of string.

Function

Use.

To extract a given number of characters from the righthand end of a given string.

Form.

RIGHT\$ (*string-expression*, *integer-expression*)

The *string-expression* gives the string from which to extract characters.

The *integer-expression* gives how many characters to extract, and must yield a value in the range 0...255.

Notes.

If the given string is shorter than the required length, then the entire string is returned. Otherwise the righthand end of the string is returned, giving a string of the required length.

Associated Keywords.

MID\$, LEFT\$

Remove a directory.

Command

Use.

For Mallard-86 MSDOS 2 version, this removes a directory from the specified, or default drive.

Mallard-86 MSDOS 1 version, all CP/M versions and Mallard-80 all ignore this command.

Form.

RMDIR *string-expression*

where the *string-expression* yields a *dir-name* which specifies the directory to remove.

Notes.

The directory must exist and be empty, except for . and ..

The directory must not be the current directory.

Associated Keywords.

CHDIR, MKDIR, CD, MD, RD, CHDIR\$, FINDDIR\$

Random Number.

Function

Use.

To get a random number. This may be the next in the current sequence, the last random number repeated or the first in a new sequence.

Form.

`RND[(numeric-expression)]`

Notes.

RND returns a Single Length value, where $0 \leq \text{value} < 1$. The pseudo-random number generator produces each number by operating on the previous one, so that from the same initial value, the same sequence is produced.

RND with the expression omitted, or with an expression yielding a value greater than zero, returns the next random number in the current sequence.

RND with an expression yielding the value zero returns a copy of the last random number generated.

RND with an expression yielding a value less than zero starts a new sequence loosely, but predictably, based on that value. The first number in the new sequence is returned.

Associated Keywords.

RANDOMIZE

ROUND

Round value.

Function

Use.

To round a value to a given number of decimal places, or to a given power of ten.

Form.

ROUND (*numeric-expression*[, *decimals*])

The *numeric-expression* yields the value to be rounded, which may be of any numeric type.

The optional *decimals* parameter specifies the number of decimal places to round to. If present *decimals* is an *integer-expression* yielding a value in the range -39...+39.

Notes.

The value of the *numeric-expression* is rounded to the number of decimal places specified by *decimals*, as follows:

decimals greater than 0

- the value is rounded to the given number of decimal digits after the decimal point.

decimals zero or absent

- the value is rounded to an integer

decimals less than zero

- the value is rounded to give ABS (*decimals*) zeros before the decimal point.

(Rounding means round to nearest as described in Section 2.9.1.)

Associated Keywords.

INT, FIX, CINT

RSET

Set one string to another, right justified.

Command

Use.

To replace the contents of one string by another, filling with spaces on the left to the same length as the original contents.

Form.

RSET *string-variable* = *string-expression*

Notes.

The *string-expression* is evaluated and then forced to the same length as the current value of the given *string-variable* – either by padding with spaces at the lefthand end, or by discarding characters from the righthand end. The result replaces the original contents of the *string-variable*.

RSET is particularly useful for setting new values into fields of a random record.

Associated Keywords.

FIELD, LSET, MID\$, MKI\$, MKS\$, MKD\$

RUN *filename*

Load and start executing a program.

Command

Use.

To load a program from file and start executing it.

Form.

RUN *string-expression*[, R]

The *string-expression* gives the filename of the file to load from. If no type extension is given .BAS is assumed.

Notes.

Any existing program, user functions, variables and arrays are deleted from memory. DEFINT, DEFSNG, DEFDBL, DEFSTR and OPTION BASE settings are reset. Unless the ,R option is specified all files are closed. RUN resets OPTION TAB, OPTION PRINT, OPTION LPRINT, OPTION INPUT and OPTION STOP to their standard states.

RUN *filename* is equivalent to LOAD *filename* followed by RUN.

RUN *filename*, R is equivalent to LOAD *filename*, R.

Associated Keywords.

LOAD, CHAIN, SAVE, CHAIN MERGE, CHAIN

RUN [*line-number*]

Start executing current program.

Command

Use.

To start executing the current program, either at the beginning, or at a given line.

Form.

RUN [*line-number*]

If the *line-number* is omitted, then execution starts at the first line of the program.

Notes.

Any existing program, user functions, variables and arrays are deleted from memory. DEFINT, DEFSNG, DEFDBL, DEFSTR and OPTION BASE settings are reset. All files are closed. RUN resets OPTION TAB, OPTION PRINT, OPTION LPRINT, OPTION INPUT and OPTION STOP to their standard states.

SAVE

Save program on disc.

Command

Use.

To write the program currently in memory to disc. Three formats for program files are supported.

Form.

`SAVE filename-expression[,form-specifier]`

The *filename-expression* gives the name of the file to which the program is to be written. If no type extension is specified, .BAS is assumed. If a file of that name already exists it is deleted.

The *form-specifier* may be one of:

- A – specifying ASCII form.
- P – specifying Protected form.

If no *form-specifier* is given Standard form is produced.

Notes.

Standard form saved files contain the BASIC program in a processed form, suitable only for reloading into memory by BASIC.

ASCII form saved files contain the BASIC program in the same form as in a listing. These files may be processed by other programs – for instance text editors.

Protected form saved files are similar to Standard form files, except that the program is encrypted. BASIC disallows any command that gives access to the program text after loading a Protected form file.

Associated Keywords.

LOAD, RUN, CHAIN, CHAIN MERGE, MERGE

SEEKKEY

Seek to a given key.

Jetsam Function

Use.

Move the current position to the first key with the given value in the given rank.

Form.

SEEKKEY (*file-reference*, *lock*, *rank*, *key-value*)

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies what lock to apply to the record referred to by the first key entry with the required *key-value* in the specified *rank*.

Return codes.

0	Success	the current position has been set to the first key entry with the required value and rank, and the associated record has been locked as specified.
103	Failure	no key entry with the required value and rank exists, nor do any higher value keys in this or any higher numbered rank exist. The current position has been unset.
105	Failure	no key entry with the required value and rank exists. The current position is set to key entry which would immediately follow the specified key if it were in the index.
132	Failure	could not read lock the record as requested.
133	Failure	could not write lock the record as requested.

Notes.

SEEKKEY may be used to access the data records randomly by key.

Associated Keywords.

SEEKRANK, SEEKREC, SEEKNEXT, SEEKSET, SEEKPREV

Seek the next key.

Jetsam Function

Use.

Starting from the given position move forward to the next key entry.

Form.

SEEKNEXT (*file-reference*, *lock*[, *index-position*])

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies the lock to apply to the next record.

The *index-position* parameter specifies which key entry to step forward from. If the parameter is omitted the current position is assumed.

Return codes.

- | | | |
|-----|---------|---|
| 0 | Success | the new current position is in the same key set (it has the same key value and rank) as the previous position. |
| 101 | Success | the new current position is not in the same key set (it has a different key value) but is in the same rank as the previous position. |
| 102 | Success | the new current position is in the next, existing, rank. |
| 103 | Failure | the current position or given <i>index-position</i> is the last key of any rank in the index. The current position has been unset. |
| 105 | Failure | the current position or given <i>index-position</i> does not exist. The current position is set to the first key entry in the key set which would immediately follow the specified position if it were in the index – if no such entry exists then the current position is unset. |
| 115 | Failure | the <i>index-position</i> parameter was omitted and the current position was unset. |
| 132 | Failure | could not read lock the record as requested. |
| 133 | Failure | could not write lock the record as requested. |

Notes.

SEEKNEXT may be used to access the data records sequentially by key.

Associated Keywords.

SEEKRANK, SEEKKEY, SEEKREC, SEEKSET, SEEKPREV

SEEKPREV

Seek the previous key.

Jetsam Function

Use.

Starting from the given position move back to the previous key entry.

Form.

SEEKPREV (*file-reference*, *lock*[, *index-position*])

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies the lock to apply to the previous record.

The *index-position* parameter specifies which key entry to step back from. If the parameter is omitted the current position is assumed.

Return codes.

0	Success	the new current position is in the same key set (it has the same key value and rank) as the old position.
101	Success	the new current position is not in the same key set (it has a different key value) but is in the same rank as the old position.
102	Success	the new current position is in the previous, existing, rank.
103	Failure	the current position or given <i>index-position</i> is the first key of any rank in the index. The current position has been unset.
105	Failure	the current position or given <i>index-position</i> does not exist. The current position is set to the first key entry in the key set which would immediately follow the specified position if it were in the index – if no such entry exists then the current position is unset.
115	Failure	the <i>index-position</i> parameter was omitted and the current position was unset.
132	Failure	could not read lock the record as requested.
133	Failure	could not write lock the record as requested.

Notes.

SEEKPREV may be used to access the data records sequentially by key.

Associated Keywords.

SEEKRANK, SEEKKEY, SEEKREC, SEEKNEXT, SEEKSET

SEEKRANK

Seek the first key of the given rank.

Jetsam Function

Use.

To move the current position to the first key entry in the given rank.

Form.

SEEKRANK (*file-reference*, *lock*, *rank*)

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies what lock to apply to the record referred to by the first key entry in the specified *rank*.

Return codes.

- | | | |
|-----|---------|--|
| 0 | Success | the current position has been set to the first key entry in the required rank, and the required record lock obtained. |
| 102 | Failure | there are no key entries in the required rank. The current position has been set to the first entry in the next existing rank. |
| 103 | Failure | there are no key entries in the required rank, nor in any higher numbered rank. The current position has been unset. |
| 132 | Failure | could not read lock the record as requested. |
| 133 | Failure | could not write lock the record as requested. |

Notes.

SEEKRANK may be used to move the current position ready for sequential processing.

Associated Keywords.

SEEKKEY, SEEKREC, SEEKNEXT, SEEKSET, SEEKPREV

SEEKREC

Set the current position.

Jetsam Function

Use.

To restore the current position from previously recorded values.

Form.

SEEKREC (*file-reference*, *lock*[, *index-position*])

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies what lock to apply to the record.

The *index-position* specifies where to set the current position. If this parameter is omitted the current position is assumed.

Return codes.

0	Success	the current position has been set as specified, and the required record lock obtained.
103	Failure	no key entry with the required value, rank and record number exists, nor do any higher value keys in this or any higher numbered rank exist. The current position has been unset.
105	Failure	no key entry with the required value, rank and record number exists. The current position is set to first key entry in the key set which would immediately follow the specified key if it were in the index.
115	Failure	the <i>index-position</i> parameter was omitted and the current position has been unset.
132	Failure	could not read lock the record as requested.
133	Failure	could not write lock the record as requested.

Notes.

SEEKREC may be used to reset the current position to a previously found key entry.

Associated Keywords.

SEEKRANK, SEEKKEY, SEEKNEXT, SEEKSET, SEEKPREV

SEEKSET

Seek the next key different from the current key.

Jetsam Function

Use.

Starting from the given position move forward to the first key entry in the next key set (ie. the next key entry with a different key value).

Form.

SEEKSET (*file-reference*, *lock*[, *index-position*])

The *file-reference* gives the file number of an open keyed file.

The *lock* parameter specifies the lock to apply to the new current record.

The *index-position* parameter specifies which key entry to step forward from. If the parameter is omitted the current position is assumed.

Return codes.

- | | | |
|-----|---------|---|
| 101 | Success | the new current position is in the same rank as the previous entry. |
| 102 | Success | the new current position is in the next, existing, rank. |
| 103 | Failure | the current position or given <i>index-position</i> is in the last key set of any rank in the index. The current position has been unset. |
| 105 | Failure | the current position or given <i>index-position</i> does not exist. The current position is set to the first key entry in the key set which would immediately follow the specified position if it were in the index – if no such entry exists then the current position is unset. |
| 115 | Failure | the <i>index-position</i> parameter was omitted and the current position was unset. |
| 132 | Failure | could not read lock the record as requested. |
| 133 | Failure | could not write lock the record as requested. |

Notes.

SEEKSET is similar to SEEKNEXT, except that the latter does not skip past key entries of equal value.

Associated Keywords.

SEEKRANK, SEEKKEY, SEEKREC, SEEKNEXT, SEEKPREV

Sign of value.

Function

Use.

To determine the sign of a given value.

Form.

SGN (*numeric-expression*)

Notes.

SGN returns an Integer value:

-1 if *numeric-expression* is < 0
0 if *numeric-expression* is = 0
+1 if *numeric-expression* is > 0

Associated Keywords.

ABS

Sine.**Function****Use.**

To calculate the Sine of a given value, given that the argument is expressed in radians.

Form.

`SIN (numeric-expression)`

The *numeric-expression* gives the angle, which must yield a value in the approximate range $-200,000 \dots 200,000$ radians.

Notes.

With angles very much greater than 2π the accuracy of this function becomes increasingly badly affected by the scaling of the angle into the range $-\pi \dots +\pi$. Rather than return a very inaccurate figure, BASIC will not evaluate SIN for values much beyond the range given above.

While the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. SIN returns a Single Length value.

Associated Keywords.

COS, TAN, ATN

SPACE\$

String of spaces.

Function

Use.

To create a string of spaces of a given length.

Form.

SPACE\$ (*integer-expression*)

where *integer-expression* must yield a value in the range 0...255, and specifies the length required. (A length of zero returns a null string.)

Associated Keywords.

SPC, TAB, STRING\$

Print a given number of spaces.

Print Function

Use.

In a print command SPC prints a given number of spaces.

Form.

SPC (*number-of-spaces*)

Number-of-spaces is an *integer-expression*. If *number-of-spaces* is negative, zero is assumed. If the value is greater than the device width then it is reduced to the range 1...*device-width* by a suitable number of subtractions of the device width. The resulting value gives the number of spaces to print.

Notes.

SPC functions may only be used as items in a print command.

The given number of spaces are printed, starting unconditionally at the current position. SPACE\$ is not quite equivalent, since BASIC would check that the string fitted on the current line, and possibly issue a carriage return before the spaces.

SPC need not be followed by a comma or semi-colon, a semi-colon is assumed (including when SPC is the last item in the print command).

Associated Keywords.

LPRINT, PRINT, PRINT #, SPACE\$, TAB

SQR

Square Root.

Function

Use.

Evaluate the square root of a given value.

Form.

`SQR (numeric-expression)`

where the *numeric-expression* must yield a value greater than or equal to zero.

Notes.

While the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. SQR returns a Single Length result.

STOP

Stop execution.

Command

Use.

To stop execution of a program, but leave BASIC in a state where the program can be restarted after the STOP command. This may be used to interrupt a program at a particular point when debugging.

Form.

STOP

Notes.

When the STOP is obeyed BASIC halts program execution and prints a 'Break' message. (In Run Only versions STOP is equivalent to END).

Execution may be restarted after a STOP by using the CONT command.

Associated Keywords.

CONT, END

String representation of numeric value.

Function

Use.

To convert given numeric value to a decimal string representation.

Form.

STR\$ (*numeric-expression*)

Notes.

The value of the *numeric-expression* is converted to a decimal string in the same form as used in a PRINT command. Note that positive values yield a string with a leading space, whereas negative values have a leading minus sign.

Associated Keywords.

VAL, PRINT, DEC\$, HEX\$, OCT\$

STRING\$

String of a particular character.

Function

Use.

To construct a string consisting of a given character repeated a given number of times. The character may be specified by its numeric value.

Form.

STRING\$ (*integer-expression*, *character-specifier*)

The *integer-expression* gives the required length of the result string, and must yield a value in the range 0...255.

The *character-specifier* may be one of:

integer-expression specifying CHR\$ (*integer-expression*)

string-expression specifying the first character of the string

Notes.

The ASCII character set is listed in Appendix VIII.

Associated Keywords.

SPACE\$

STRIP\$

Strip Bit 7 from all characters in a string.

Function

Use.

To create a string which is a copy of another, except that Bit 7 of every character in the string is set to zero.

Form.

STRIP\$ (*string-expression*)

Notes.

The result of the string expression is returned, with every character forced into the normal ASCII range (values 0...127) by setting the top bit (Bit 7) to zero. This may be used to remove any parity setting on characters, or to remove markers which make use of the 'extra' bit.

Associated Keywords.

LOWER\$, UPPER\$

SWAP

Swap the contents of two variables.

Command

Use.

To exchange the contents of two variables, without requiring an intermediate variable.

Form.

SWAP variable, variable

The two *variables* must be of the same type.

SYSTEM

Return to system level.

Command

Use.

To leave BASIC and return to system level.

Form.

SYSTEM

Notes.

All open files are closed.

Space forward to given position.

Print Function

Use.

In a print command TAB prints spaces to move to the given print position.

Form.

TAB (*print-position*)

print-position is an *integer-expression*. If *print-position* is less than one, one is assumed. If the value is greater than the device width, then it is reduced to the range 1...*device-width* by a suitable number of subtractions of the device width. The resulting value gives the print position to which to move.

Notes.

TAB functions may only appear as items in a print command.

If the required print position is greater than or equal to the current position, spaces are printed until the required position is reached (this may print nothing at all).

If the required print position is less than the current position, BASIC issues a new line followed by spaces to reach the required position on the new line.

TAB need not be followed by a comma or semi-colon; a semi-colon is assumed (including when TAB is the last item in the print command).

Associated Keywords.

LPRINT, PRINT, PRINT #, SPC

Tangent.

Function

Use.

To calculate the Tangent of a given value, given that the argument is expressed in radians.

Form.

TAN (*numeric-expression*)

The *numeric-expression* gives the angle in radians, and must yield a value in the approximate range $-200,000$ to $+200,000$.

Notes.

With angles very much greater than 2π the accuracy of this function becomes increasingly badly affected by the scaling of the angle into the range $-\pi \dots +\pi$. Rather than return a very inaccurate figure, BASIC will not evaluate TAN for values much beyond the range given above.

While the *numeric-expression* may be of any numeric type, the argument is forced to Single Length before it is processed. TAN returns a Single Length value.

Associated Keywords.

COS, SIN, ATN

TRON TROFF

Turn on and off tracing.

Command

Unavailable in Run Only versions

Use.

BASIC includes a facility to trace the execution of a program, by reporting the number of each line just before it is executed.

Form.

TRON

TROFF

Notes.

TRON enables the trace feature. TROFF disables it.

While tracing is enabled the number of each line is printed on the console (enclosed in square brackets) just before the line is executed.

TRON is automatically reset whenever a new program is loaded, that is by NEW, LOAD, CHAIN and RUN 'file' commands.

TYPE

Type file to console.

Command

Use.

To print a given file on the console.

Form.

`TYPE filename`

The filename may not contain 'wild card' characters.

Notes.

The file specified is read and written directly to the console. Typing Control-C will abandon the operation and return to Direct Mode (except in Run-Only versions or after OPTION RUN, when Control-C is ignored).

While the file is being written to the console, typing Control-S will suspend output (except in Run-Only versions or after OPTION RUN, when Control-S is ignored). Output may be restarted in the usual way (see Section 3.3).

TYPE takes the rest of the current line as its argument, irrespective of colons or single quotes.

Tabs and carriage returns are treated in the usual way, and console width is enforced. All other characters are sent to the console exactly as read from the file, so control characters and characters outside the normal ASCII range may be included in the file to produce special console effects.

Associated Keywords.

DISPLAY

Convert unsigned integer.

Function

Use.

To convert the given value to an Integer, interpreting the value as an unsigned sixteen bit integer.

Form.

UNT (*address-expression*)

Notes.

UNT returns an Integer value in the range -32768 to $+32767$ (ie. the 2's complement equivalent of the unsigned value of the *address-expression*). See Section 2.10 for a description of unsigned integers.

Associated Keywords.

INT, FIX, CINT, ROUND

UPPER\$

Convert string to upper case.

Function

Use.

To create a new string which is a copy of another, with all lower case alphabetic characters converted to upper case equivalents.

Form.

UPPER\$ (*string-expression*)

Notes.

The result of the *string-expression* is returned, with any characters in the range a...z converted to the equivalent character in the range A...Z.

Associated Keywords.

LOWER\$

Invoke external user function.

Function

Use.

Invokes one of the ten possible external user functions. External functions are defined by DEF USR commands, in which the machine address of the function is given.

Form.

USR[*digit*] (*expression*)

The *digit* may be in the range 0...9. If *digit* is omitted, 0 is assumed. USR *digit* must previously have been defined in a DEF USR command.

The *expression* may be any form of expression.

Notes.

See Appendix III for a discussion of external functions and subroutines.

Associated Keywords.

DEF USR, CALL, DEF SEG

Convert string to numeric value. Function

Use.

To take a string representation of a number and convert it to a numeric value.

Form.

VAL (*string-expression*)

Notes.

VAL converts the given string to a number in the same way as INPUT. The result returned will either be Integer or Double Length (so if there is any fraction part VAL returns the maximum possible precision).

Associated Keywords.

STR\$

Get pointer to variable.

Function

Use.

To get the address of a given variable or file buffer, so that it can be passed to an external function or subroutine.

Form.

VARPTR (*variable*)

or VARPTR (*#file-number-expression*)

The first form returns the address of the given variable. If the variable is subscripted, the address of that element of the array is returned.

The second form returns the address of the buffer for the given file. The file must be open.

Notes.

VARPTR returns a Single Length value in the range 0...65535.

See Appendix III for a discussion of external functions and subroutines.

When used with a JETSAM keyed file, VARPTR returns the address of the first reserved byte; 2 should be added to this value to give the address of the first byte of user data.

Associated Keywords.

CALL, USR, DEF USR, UNT

VERSION

Determine which version of BASIC is in use.

Function

Use.

BASIC makes every effort to hide the differences which exist between the various processors and operating systems with which it may be used. Occasionally a program may need to know about the version of BASIC it is running under. This function may be used to get this information.

Form.

VERSION (*integer-expression*)

The *integer-expression* must yield a value in the range 0...4, specifying what type of version information is required.

Notes.

The VERSION function returns an integer value. The meaning of that value depends on the parameter as follows:

Parameter	Return Values	Meaning
0	7	8080/8085 version of BASIC
	8	Z80 version of BASIC
	16	8088/8086 version of BASIC
1	0	Full BASIC interpreter
	1	Run-Only BASIC Interpreter
2	1	Single User BASIC
	2	Multiuser BASIC
3	1	CP/M type BASIC
	2	MS-DOS (PC-DOS)type BASIC
4	*	Operating System dependent version information

Wait on I/O port.

Command

Use.

To wait until a given I/O port returns a particular bit pattern.

Form.

WAIT *port-number*, *mask* [, *inversion*]

WAITW *port-number*, *mask* [, *inversion*]

For Mallard-80 the *port-number* is an *integer-expression*, which must yield a value in the range 0..255.

For Mallard-86 the *port-number* is an *address-expression*.

In WAIT both the *mask* and *inversion* parts are *integer-expressions*, which must yield values in the range 0..255.

In WAITW both the *mask* and *inversion* parts are *address-expressions*.

Notes.

BASIC enters a loop reading from the I/O port, Exclusive ORing the value read with the *inversion* and then ANDing with the *mask*, until the result is non zero. (If no *inversion* is given, then the Exclusive OR step is omitted.) WAIT reads bytes from the I/O port; WAITW reads words.

Note that there is no way of interrupting this loop, so BASIC will remain in it indefinitely if the required condition never occurs.

Associated Keywords.

INP, INPW, OUT, OUTW

WEND

Mark the end of a WHILE loop.

Command

Use.

A WHILE loop repeatedly executes a section of a program while a given condition is true. The WEND command defines the end of the loop.

Form.

WEND

Notes.

The WEND command defines the end of a WHILE loop. The way in which WHILE and WEND are tied together is described under WHILE. When a WEND command is encountered BASIC knows which (if any) WHILE it must be associated with.

It is an error to execute a WEND which is not associated with an active WHILE.

Associated Keywords.

WHILE

WHILE

Start a WHILE loop.

Command

Use.

A WHILE loop repeatedly executes a section of program while a given condition is true. The WHILE command defines the head of a loop, and gives the condition. The WEND command defines the end of a loop.

Form.

WHILE *logical-expression*

Notes.

When the WHILE command is obeyed the *logical-expression* is evaluated. If the result is zero BASIC skips to the matching WEND. If the result is not zero, execution continues until the matching WEND is met, whereupon BASIC loops back to the WHILE command, and the process is repeated.

The WHILE loop does not terminate suddenly when the condition given by the *logical-expression* becomes false. The condition is only tested once each time the end of the loop, the WEND, is reached.

The WEND command which matches a given WHILE is established statically when the WHILE is first executed. That is to say that the WEND which matches the WHILE depends on the order of statements in the program, quite independent of the order of execution. It is not possible, therefore, to have more than one WEND associated with a given WHILE.

WHILE loops may be nested, both inside other WHILE loops and inside FOR loops.

It is permissible to terminate a WHILE loop by avoiding the WEND.

Associated Keywords.

WEND, FOR

WIDTH

Set width of console.

Command

Use.

To tell BASIC how wide the console is in characters. This information is used when printing to the console, to allow BASIC to generate new lines at the appropriate moments.

Since many consoles automatically start new lines, WIDTH may be used to tell BASIC where the console does this, so that it can avoid generating new lines as well.

Form.

WIDTH *width*[, *Auto-CRLF-column*]

Width is an *integer-expression* which gives the console width in characters, and must yield a value in the range 1 to 255 (though small values may have curious effects).

Auto-CRLF-column, if given, is an *integer-expression* which gives the console's Auto CRLF position (see below), and must yield a value in the range 1..255. If the parameter is omitted the previous setting is retained.

Notes.

Setting the width to 255 has special meaning. BASIC treats the console as being infinitely wide, so never generates new lines. BASIC maintains a counter giving the logical position on the console. When the logical position reaches 255 the counter is no longer incremented, so all positions greater than 255 are treated as 255. The line editor is not affected by setting width 255, but continues to operate with the last 'real' (ie non-255) width setting.

Setting the Auto CRLF column to 255 causes BASIC to assume that the console never generates new lines. Any other setting gives the column which when written to causes the console to generate a new line. BASIC will avoid generating an extra new line when the width and Auto CRLF column are equal. It is not sensible to set the Auto CRLF position less than the screen width (except when special width 255 is used).

The width setting affects all output to the console.

Associated Keywords.

WIDTH LPRINT, PRINT, POS, EDIT

WIDTH LPRINT

Set width of line printer.

Command

Use.

To tell BASIC how wide the line printer is in characters. This information is used when printing to the line printer, to allow BASIC to generate new lines at the appropriate moments.

Form.

WIDTH LPRINT *integer-expression*

The *integer-expression* must yield a value in the range 1 to 255 (though small values may have curious effects).

Notes.

The initial value of line printer width is 132.

Setting the width to 255 has special meaning. BASIC treats the line printer as being infinitely wide, so never generates new lines. BASIC maintains a counter giving the logical position on the printer. When the logical position reaches 255 the counter is no longer incremented, so all positions greater than 255 are treated as 255.

The width setting affects all output to the line printer.

Associated Keywords.

WIDTH, LPRINT, LPOS

WRITE

Write to console.

Command

Use.

To print the values of a number of expressions to the console, separating them by commas and enclosing strings in double quotes.

Form.

WRITE [*write-list*]

write-list is: *expression*[*separator expression*]*

where *separator* may be comma or semi-colon, interchangeably.

(The [...]* construct indicates an optional part, which, if present, may be repeated any number of times.)

Notes.

Write is similar to PRINT, except that:

- print zones are ignored
- strings are printed enclosed in double quotes
- commas are added between the printed items
- WRITE does not support the trailing separator option

Associated Keywords.

WRITE #, PRINT

WRITE

Write to file.

Command

Use.

To print the values of a number of expressions to the given file separating them by commas and enclosing strings in double quotes. WRITE # puts data to the file in a form that INPUT # is able to read back.

Form.

WRITE #*file-number-expression*, [*write-list*]

The *file-number-expression* specifies the file to write to. The file must be open for Output or Random Access.

write-list is: *expression*[*separator expression*]*

where *separator* may be comma or semi-colon, interchangeably.

(The [..]* construct indicates an optional part, which, if present, may be repeated any number of times.)

Notes.

WRITE # is similar to PRINT # except that:

- print zones are ignored
- strings are printed enclosed in double quotes
- commas are added between the printed items
- WRITE # does not support the trailing separator option
- if the file is open for Random Access the record is space filled to the *record-length-1* before the carriage return is inserted.

Associated Keywords.

WRITE, PRINT #

ZONE

Set Print Zone Size.

Command

Use.

To change the width of the Print Zone used in PRINT, LPRINT and PRINT #.

Form.

ZONE *integer-expression*

The *integer-expression* gives the new Print Zone width, and must yield a value in the range 1 to 255.

Notes.

The default Print Zone width is 15. The width is reset to the default value whenever a new program is loaded, that is by NEW, LOAD, CHAIN and RUN 'file' commands.

Associated Keywords.

PRINT, LPRINT, PRINT #, WIDTH

Initialising BASIC

When you load BASIC, the command line may include a number of parameters, as follows:

[file-name][/F : number-of-files][/M : address][/S : size]

If no parameters are given, BASIC is initialised using default values and enters Direct Mode.

If the *file-name* parameter is given, it must be the first parameter. When BASIC has completed initialisation, the given file is loaded and execution begins immediately.

The other parameters may be given in any order.

/F : BASIC can handle a limited number of files at once, /F sets this number. The *number-of-files* may be in the range 0...255 (though it may prove difficult to find enough memory for more than a hundred files). If no /F: is specified, the maximum is set to 3.

Each file requires approximately 176 bytes of memory plus the buffer specified by the /S: setting, unless that buffer is less than 128 bytes, when a total of 304 bytes are required.

/M : Sets the limit on BASIC's memory use. The *address* gives the address of the last (highest address) byte of memory which may be used by BASIC. If no /M: is specified BASIC uses as much memory as it can:

Mallard-80 takes the value at locations 6 and 7 as the limit of its memory use.

Mallard-86 attempts to get a full 64K byte data area from the host operating system. Some of this area is used for BASIC's variables. If a /M: limit is specified then any area beyond it in the data segment will not be used by BASIC.

BASIC requires at least 1024 bytes of free space to run at all.

/S : Sets the maximum size of random records. The *size* is expressed in bytes. If no /S: is specified, 128 is assumed.

Appendix I: Initialising BASIC

The various numbers may be unsigned decimal integers (maximum value 65535), or may be entered in Octal or Hexadecimal (using &O and &H notation).

These values may be altered after BASIC is loaded by using the MEMORY and CLEAR commands.

Error Numbers and Error Messages

All errors generated by BASIC are listed here, in error number order. The messages produced by BASIC are given, as well a brief description of possible causes.

1. Ordinary BASIC Errors

1 `Unexpected NEXT`

A NEXT command has been encountered while not in a FOR loop, or the control variable in the NEXT command does not match that in the FOR.

2 `Syntax Error`

3 `Unexpected RETURN`

A RETURN command has been encountered when not in a subroutine.

4 `DATA exhausted`

A READ command has attempted to read beyond the end of the last DATA.

5 `Improper argument`

This is a general purpose error. The value of a function's argument, or a command parameter is invalid in some way.

6 `Overflow`

The result of an arithmetic operation has overflowed. This may be a floating point overflow, in which case some operation has yielded a value greater than 1.7E+38 (approx.). Alternatively, this may be the result of a failed attempt to change a floating point number to a 16 bit signed integer.

Appendix II: Error Numbers and Error Messages

- 7 **Memory full**
- The current program or its variables may be simply too big. If the control structure is very deeply nested (nested GOSUBs, WHILEs or FORs) then the stack may be too small – the CLEAR or MEMORY commands allow more stack to be specified.
- If this occurs while editing CLEAR may be able to make more room by discarding all variables.
- If this occurs when attempting to lock a record the BUFFERS command may be used to increase the maximum number of record locks per file. Note that the default maximum is zero.
- 8 **Line does not exist**
- The line referenced cannot be found.
- 9 **Subscript out of range**
- One of the subscripts in an array reference is too big or too small.
- 10 **Array already dimensioned**
- One of the arrays in a DIM statement has already been declared, or an OPTION BASE command has been issued again or too late.
- 11 **Division by zero**
- May occur in floating point division, integer division, integer modulus or in exponentiation.
- 12 **Invalid direct command**
- The last command attempted is not valid in Direct Mode.
- 13 **Type mismatch**
- A numeric value has been presented where a string value is required, and vice versa, or an invalidly formed number has been found in READ or INPUT #.
- 14 **String space full**
- So many strings have been created that there is no further room available, even after 'garbage collection'.
- 15 **String too long**
- String exceeds 255 characters in length. May be generated by concatenating strings.

- 16 `String expression too complex`
String expressions may generate a number of intermediate string values. When the number of these values exceeds a reasonable number, BASIC gives up and this error results.
- 17 `Cannot CONTinue`
For one reason or another the current program cannot be restarted using `CONT`. Note that `CONT` is intended for restarting after a `STOP` command, `Control-C` or error, and that any alteration of the program in the meantime makes a restart impossible.
- 18 `Unknown user function`
No `DEF FN` has been executed for the `FN` just invoked.
- 19 `RESUME missing`
The end of the program has been encountered while in Error Processing Mode (ie. in an `ON ERROR GOTO` routine).
- 20 `Unexpected RESUME`
`RESUME` is only valid while in Error Processing Mode (ie. in an `ON ERROR GOTO` routine).
- 21 `O/S dependent error`
The `OSERR` function will return an operating system dependent value which will further identify the error.
- 22 `Operand missing`
BASIC has encountered an incomplete expression.
- 23 `Line too long`
- 26 `NEXT missing`
BASIC cannot find a `NEXT` to match a `FOR` command.
- 29 `WEND missing`
BASIC cannot find a `WEND` to match a `WHILE` command.
- 30 `Unexpected WEND`
BASIC has encountered a `WEND` when not in a `WHILE` loop, or a `WEND` that does not match the current `WHILE` loop.

2. Disc and File Related Errors

- 50 Record overflow
Run out of record in a FIELD command, or any command reading or writing the random record buffer.
- 51 Internal error
Any occurrence of this error should be reported.
- 52 File number error
An out of range file number has been specified, or an attempt has been made to read or write a file which is not open.
- 53 File not found
Unable to open the required file.
- 54 File type error
A file operation has been attempted which is not consistent with the way in which the file was opened. This error also occurs when BASIC is reading what it expects is a program file, but cannot recognise the contents.
- 55 File already open
The file number specified is in use.
- 57 Disc I/O error
- 58 File already exists
A file with the new name in a NAME or REN command already exists.
The CREATE Jetsam command will report this error if either the index or the data file already exists.
- 61 Disc full
- 62 EOF met
An attempt has been made to read past the end of a sequential file.

- 63 Record number error
A record number outside the legal range 1....32767 has been specified, or an impossible record number has been used when accessing a keyed data file.
- 64 File name invalid
- 66 Direct command found
When loading a program file BASIC has encountered a direct command. May indicate that the file is an ASCII file generated outside BASIC and which is not in a suitable form.
- 67 Directory full
- 68 Read past EOF
Can occur on some operating systems if an attempt is made to lock a record which has never been written to.
- 69 Record is locked
A GET or PUT operation on a random file record has failed because the record is already locked and the existing lock is incompatible with the required new lock.
- 70 Read-only disc
It is not possible to write to the disc in question.
- 71 Read-only file
It is not possible to write to the file in question.
- 72 Invalid drive
The drive in question does not exist.
- 73 File is locked
A file lock operation has failed because the file is already locked and the existing lock is incompatible with the required new lock.
- 74 RESET denied
The operating system has rejected a disc reset command.

3. Jetsam Specific Error Numbers

The following errors are specific to Jetsam:

- 113 Not a keyed file
- When OPENing a keyed file Jetsam has not recognised the header records of either the data and or the index file.
- 114 Record is not locked
- An attempt has been made to GET or PUT a record which is not adequately locked.
- 115 Inconsistent files
- When OPENing a keyed file Jetsam has found that the data and index files are marked inconsistent. This indicates that for some reason the keyed file was not CLOSED or CONSOLIDATED after it was last written to. The keyed file is not usable, and the user should turn to the latest back-up copy.

External Routines

Commands exist to invoke external machine language routines. Two forms of routine are supported – **USR Functions** and **CALLED Subroutines**.

BASIC requires external routines to be loaded into machine memory outside the area used by the **BASIC**. With **Mallard-80** this is typically above the highest address used by **BASIC**. **Mallard-86** allows the routines to be located in their own segment, or to use memory in **BASIC**'s data segment above the highest address used by **BASIC**. The highest address used by **BASIC** may be set by the **/M:** load time option (see Appendix I) or by the **CLEAR** and **MEMORY** commands.

1. Data Formats

This is a brief description of the format of the four data types. All are held with the least significant byte first (at the lowest address).

Integer. Two's complement 16 bit values.

String. Strings are held in two parts, the **String Descriptor** and the **String Body**. The **String Descriptor** is a three byte vector. The least significant byte gives the length of the string, zero implies the string is null and that the rest of the descriptor should be ignored. The next two bytes give the machine address of the string body. (In **Mallard-86** the body of the string is in **BASIC**'s Data Segment).

Single Length. A three byte mantissa followed by a one byte exponent. Numbers are always held normalised, and the most significant bit of the mantissa is replaced by the sign. The exponent is biased by 128. A zero exponent means that the number is zero, and the mantissa should be ignored. The mantissa is in sign and magnitude form with the binary point to the left of the implied most significant bit. The first byte of the mantissa is the least significant byte.

Double Length. As single length, but with four more bytes of mantissa.

2. USR Functions

Ten names are reserved for external functions. They are USR0 to USR9. The DEF USR command declares the address of an external function and assigns it to the given USR name. USR functions are invoked in the usual way and take one parameter, which may be of any type. The function must return a value, which may either be of the same type as the parameter or of Integer type.

The entry conditions of the function are as follows:

Mallard-80: the A register indicates the type of the parameter:

- 2 => Integer Parameter
HL register pair contains the address of a 16 bit 2's complement value.
- 3 => String Parameter
DE register pair contains the address of the string descriptor (as described above).
- 4 => Single Length Parameter
HL register pair contains the address of the four byte single length number.
- 8 => Double Length Parameter
HL register pair contains the address of byte 4 of the eight byte double length number (where the first byte is byte 0).

Mallard-86: the AL register indicates the type of the parameter:

- 2 => Integer Parameter
BX register pair contains the address of a 16 bit 2's complement value.
- 3 => String Parameter
DX register pair contains the address of the string descriptor (as described above).
- 4 => Single Length Parameter
BX register pair contains the address of the four byte single length number.

8 => Double Length Parameter

BX register pair contains the address of byte 4 of the eight byte double length number (where the first byte is byte 0).

All addresses refer to locations in BASIC's Data Segment. See below for a description of the segment register settings.

The state of other registers (except segment registers) is undefined.

Note that there is limited stack available, so functions with large stack requirements should have their own stack area.

The exit conditions are:

All registers and flags, except segment registers, may be corrupted.

Either: A value of the same type as the original returned in the area occupied by the parameter.

Or: An Integer value returned using the RETURN_INTEGER subroutine (see below).

While USR functions may take a String parameter and return a String value, the function may not alter the String Descriptor in any way.

It is expected that USR functions will, in general, require Integer parameters and return Integer values. Two routines are provided by BASIC to support this pattern of use. They are:

GET_INTEGER

Force the parameter to an Integer, rounding floating point numbers. Fails if the parameter is a string. (Equivalent to CINT).

Entry conditions:

The parameter has not been altered.

Exit conditions:

Mallard-80: HL contains the Integer value of the parameter

Mallard-86: BX contains the Integer value of the parameter

Other registers (except for segment registers) and flags corrupt

RETURN_INTEGER

Set the value to be returned by the function to be the Integer given.

Entry conditions:

Mallard-80: HL contains the Integer value to be returned

Mallard-86: BX contains the Integer value to be returned

Exit conditions:

Mallard-80: register A and flags corrupt

Mallard-80: register AL and flags corrupt

Other registers preserved

In Mallard-80 the addresses of these two routines appear within the BASIC interpreter at the following locations:

GET_INTEGER 259 Decimal 103 Hexadecimal

RETURN_INTEGER: 261 Decimal 105 Hexadecimal

In Mallard-86 the routines may be invoked by a FAR CALL to the following locations in the BASIC interpreter:

GET_INTEGER 259 Decimal 103 Hexadecimal

RETURN_INTEGER: 263 Decimal 107 Hexadecimal

3. CALLED Subroutines

The CALL command specifies the address of the subroutine, optionally followed by a list of parameters. The address may only be specified by the value of a variable – ie. not by an expression. The parameters may only be variables – they may not be expressions. (Note that this includes variables which are array items.)

Parameters are passed by reference, that is the address of the value of each parameter is passed (in the case of a String, the value is the String Descriptor). The number and type of parameters must be agreed between the BASIC program and the subroutine – there is no checking. If possible the addresses are passed in registers, otherwise they are passed in an area of memory.

For Mallard-80 the entry conditions are:

If there are 3 or fewer parameters:

HL contains the address of parameter 1 (if any)

DE contains the address of parameter 2 (if any)

BC contains the address of parameter 3 (if any)

If there are 4 or more parameters:

HL	contains the address of parameter 1		
DE	contains the address of parameter 2		
BC	contains the address of an area of memory containing the addresses of the other parameters, thus:		
	2 byte	address of parameter n	high address
		
		address of parameter 4	
		address of parameter 3	low address
BC	contains the address of the less significant byte of the address of parameter 3.		

For Mallard-86 the entry conditions are:

Stack:	2 byte	address of parameter 1	high address
		address of parameter 2	
		
		address of parameter n	
		BASIC's Code Segment	
		return offset	low address

The SP register points at the less significant byte of the return offset. Parameters may be referenced using suitable BP relative addressing modes. All addresses refer to locations in BASIC's Data Segment.

See below for a description of the segment register state.

The contents of other registers (except segment registers) are undefined.

The exit conditions are:

All registers (except segment registers) and flats corrupt.

Since parameters are passed by reference it is possible for the subroutine to return values. Access to variables is as described previously, for USR routines. New String values may be returned provided that the subroutine does **not** alter the String Descriptor in any way.

4. Mallard-86 and Segment Registers

When either a `USR` or a `CALLED` external routine is called Mallard-86 uses a `FAR CALL`. The offset of the address called is given by the appropriate `DEF USR` and by the value of the variable in the `CALL` statement. The segment base used is that last set by `DEF SEG` (the default segment is BASIC's data segment). When the routine is entered the value at the top of stack is the 'far return' to BASIC. The segment registers are set up:

CS: as set in the latest `DEF SEG` (or BASIC's data segment if none)

SS: BASIC's data segment

DS: BASIC's data segment

ES: BASIC's data segment

All addresses of parameters are offsets within BASIC's data segment. To use either the `GET_INTEGER` or the `RETURN_INTEGER` the routine must create a 'far address' by picking BASIC's code segment from the stack and using the offset given above.

If any segment registers are modified in an external routine they **MUST** be restored to their entry values before the routine returns.

Appendix IV

BASIC Keywords

The following are the BASIC keywords. They are reserved and cannot be used as variable names.

ABS, ADDKEY, ADDREC, ALL, AND, AS, ASC, ATN, AUTO

BASE, BUFFERS

CALL, CD, CDBL, CHAIN, CHDIR, CHDIR\$, CHR\$, CINT, CLEAR, CLOSE, COMMON, CONSOLIDATE, CONT, COS, CREATE, CSNG, CVD, CVI, CVIK, CVS, CVUK

DATA, DEC\$, DEF, DEFDBL, DEFINT, DEF SEG, DEFSNG, DEFSTR, DEL, DELETE, DELKEY, DIM, DIR, DISPLAY

EDIT, ELSE, END, EOF, EQV, ERA, ERASE, ERL, ERR, ERROR, EXP

FETCHKEY\$, FETCHRANK, FETCHREC, FIELD, FILES, FIND\$, FINDDIR\$, FIX, FN, FOR, FRE

GET, GOSUB, GOTO

HEX\$, HIMEM

IF, IMP, INKEY\$, INP, INPUT, INPUT #, INPUT\$, INPW, INSTR, INT

KILL

LEFT\$, LEN, LET, LINE, LIST, LLIST, LOAD, LOC, LOCK, LOF, LOG, LOG10, LOWER\$, LPOS, LPRINT, LSET

MAX, MD, MEMORY, MERGE, MIDS\$, MIN, MKD\$, MKDIR, MKI\$, MKIK\$, MKS\$, MKUK\$, MOD

NAME, NEXT, NEW, NOT

OCT\$, ON, ON ERROR GOTO 0, OPEN, OPTION, OR, OSERR, OUT, OUTW PEEK, POKE, POS, PRINT, PRINT #, PUT

RANDOMIZE, RANKSPEC, RD, READ, REM, REN, RENUM, RESET, RESTORE, RESUME, RESUME 0, RETURN, RIGHT\$, RMDIR, RND, ROUND, RSET, RUN

Appendix IV: BASIC Keywords

SAVE, SEEKKEY, SEEKNEXT, SEEKPREV, SEEKRANK, SEEKREC,
SEEKSET, SGN, SIN, SPACE\$, SPC, SQR, STEP, STOP, STR\$, STRING\$,
STRIP\$, SWAP, SYSTEM

TAB, TAN, THEN, TO, TROFF, TRON, TYPE

UNT, UPPER\$, USING, USR

VAL, VARPTR, VERSION

WAIT, WAITW, WEND, WHILE, WIDTH, WRITE, WRITE #

XOR

ZONE

Hints on the Use of Jetsam and Multiuser Facilities

1. Numeric Keys

Jetsam keys are strings. DEC\$ may be used to create a suitable ASCII string for numeric keys. Jetsam works faster with short keys than it does with longer keys. The expansion of numeric keys to ASCII strings should be avoided if possible. For integral values, the following are recommended:

- i. for values in the range 0...255.

Use CHR\$(x%) to produce a 1 byte string.

- ii. for values in the range -32768 to + 32767 (ie. Integer)

Use MKIK\$(x%) to produce a 2 byte string.

- iii. for values in the range 0...65535 (ie. Unsigned Integer)

Use MKUK\$(x) to produce a 2 byte string.

It is well worthwhile transforming numeric key values into one of the above integer ranges to minimise key sizes.

If fractional key values must be used then DEC\$ is the only practical solution (but note that negative numbers require special treatment). Neither of the functions MKD\$ and MKS\$ produce strings of a suitable form.

2. Sequential Processing of Keyed Files in a Multiuser Environment

The following fragment of code will step on to the next record and lock it while processing a file sequentially:

```
10 previous.records% = FETCHREC(#file.number%)
20 return.code% = SEEKNEXT(#file.number%, 1)
30 IF previous.record%<>0 THEN IF LOCK(#file.number%, 0,
                                previous.record% THEN STOP
40 IF return.code%
```

The procedure is to find the current record number; step to the next record and (read) lock it: and then unlock the old current record (if the position was set) – this operation must succeed so STOP is a suitable action if it fails!

Assuming the current record is locked when the fragment is entered, this procedure will avoid any problems with the current position being deleted because the move to the next record and lock is done before unlocking the old position.

Alternatively the following:

```
10 IF FETCHREC(#file.number%)<>0 THEN IF
    LOCK(#file.number%, 0, FETCHREC(#file.number%)) THEN STOP
20 return.code% = SEEKNEXT(#file.number%, 1)
30 IF return.code%
```

will also work, but if the old position is deleted between unlocking it and moving to the next record, a return code of 105 will be generated by the SEEKNEXT.

3. Sub-Keys

Some sorting systems work with a primary key and a variable number of sub-keys. If records with equal primary keys are found then the first sub-keys are used to establish the ordering. If the first sub-keys are also equal then the second are used, and so on. Jetsam does not support such a scheme. The ranks in Jetsam do not constitute sub-keys, the keys in one rank are quite independent of keys in any other rank.

To achieve the same effect as a primary and sub-keys system the program must create a single Jetsam key by concatenating the required application level keys. This is straightforward if the application's keys are fixed length strings, for example:

```
10 IF ADDREC #file.number%,0,rank%,  
    MKIK$(date%)+MKIK$(invoice%)) THEN PRINT "ADDREC FAILED"
```

will insert a new record which is sorted into the index by date and invoice number. If the application's keys are not fixed length they may be separated by a CHR\$(0), for example:

```
10 IF ADDREC (#file.number%,0,rank%,  
    surname$+CHR$(0)+initials$) THEN PRINT "ADDREC FAILED"
```

will insert a new record which is sorted into the index file by surname and initials.

4. Straightforward Multiuser Random File Handling

The Temporary Record Write Lock described in Section 8.3.4 is designed to simplify conversion of existing random file processing programs. Programs which read one record at a time, process it and, perhaps, then write it back do not require any explicit record locking, because BASIC automatically applies a write lock. Only the OPEN need be modified, to open the file unlocked, because the default is to open it write locked.

If this mechanism is used it is important to recognise its limitation, which is that only one record, the last one read, is locked. Programs which require the contents of more than one record to remain unchanged while one or more of them is processed will require explicit locks on all the records in question.

5. Guard Record

A dummy, or guard, record may be used by the application to control access to the file in ways not provided by the built-in record and file lock mechanisms. The following example is for a keyed file, but the technique applies equally to random files.

Appendix V: Hints on the Use of Jetsam and Multiuser Facilities

To control access to a complete rank of keys, the application could create a dummy record in the rank and use the following fragments of code:

```
10 REM To gain non-exclusive access to a rank
20 return.code% = SEEKKEY(#file.number%,1,rank%,")
30 IF return.code%<>0 THEN
.
.
90 IF LOCK(#file.number%,0,FETCHREC(#file.number%))
THEN STOP
```

and:

```
10 REM To gain exclusive access to a rank.
20 return.code%=SEEKKEY(#file.number%,2,rank%,")
30 IF return.code%<>0 THEN
.
.
90 IF LOCK(#file.number%,0,FETCHREC(#file.number%))
THEN STOP
```

The successful application of a read lock in the first fragment allows other shared access to the rank for which the record is the guard. Similarly the application of the write lock in the second fragment gains exclusive access.

A guard record with similar fragments of code may be used to control access to any part or all of a random or a keyed file.

Installation of BASIC

The Installation procedure allows the following to be set:

- a. Whether to use the screen based or the command driven line editor.
- b. The screen control characters and sequences for the screen based line editor.
- c. The keyboard control characters and sequences for the screen based line editor.
- d. The console width and Auto CRLF position.
- e. The space compression feature (see Section 2.2).

By default, the command line editor is enabled, the console width and Auto CRLF positions are both set to 80, and the space compression feature is enabled.

Full BASICs include a "screen based line editor". This presents the line being edited on one or more screen lines. The user is allowed to move the cursor within the line and add or remove characters. The editor updates the display each time the line is changed so that the latest version is always visible.

To achieve the screen based editing the BASIC used a small number of screen cursor movement commands and needs some knowledge of the screen with which it is being used. BASIC contains a table with this information in it. The keystroke sequences for the editor commands are not fixed, but are also contained in a table in BASIC.

The Install program is provided for the user to set up these tables so that BASIC may be made compatible with the screen and keyboard with which it is to be used. Some versions of BASIC are supplied fully installed for a particular computer. In these versions the installation program is not supplied.

Versions of Mallard BASIC supplied specifically for an IBM PC (or PC compatible) or for one of the Amstrad range of computers are fully installed for use on these machines.

Run Only BASICs do not include the line editor, and do not require any installation, except to set the console width and Auto CRLF position, if these are not both 80.

1. Screen Requirements

The editor requires a screen with the following cursor facilities:

- Move the cursor one position to the left (not character delete).
- Move the cursor one position to the right.
- Move the cursor down one line, scroll up if necessary.
- Move the cursor up one line.
- Move the cursor to the start of the current screen line.

In general BASIC expects to invoke the cursor commands by sending a control character with up to four parameter characters. The form of these control character sequences is set into BASIC's tables by the Install program. On the IBM PC, PC-DOS tends not to support control codes of this sort so the screen must be driven directly. The Install program includes an option to enable an IBM PC screen driver.

The editor also needs to know how wide the screen is and what effect writing a character to the last column of the screen has. Many screens automatically perform line feed and carriage return (Auto CRLF) after a character is written to the last column. The WIDTH command may be used to set the width of the screen and to declare the column at which Auto CRLF occurs (if any).

2. Keyboard Commands for the Editor

The commands obeyed by the Editor are described in Chapter 4. Each command may be invoked by a unique sequence of up to three characters. It is expected that the first character of a sequence will be a control character (value &H7F or in the range &H00...&H1F). The commands and a possible set up are as follows:

Cursor Left	Control-S
Cursor Right	Control-D
Cursor Up	Control-E
Cursor Down	Control-X
Find Character	Control-L
Insert/Overstrike	Toggle Control-V
Forwards delete	Control-G
Backwards delete	Rub Out (value &H7F)
Delete to character	Control-K
Complete the Edit	Carriage Return (Control-M)
Abandon editing	Control-C

3. The Install Program

The Install program is written in BASIC and may be run by the Mallard BASIC with which it is supplied. The program is intended to be straightforward to use and includes a certain amount of help information. Any machine running Mallard BASIC can install Mallard BASIC for any other machine. The Install program writes into BASIC's tables the following:

- The default screen width and Auto CRLF position.
- Whether or not the editor is enabled.
- The screen cursor command control sequences or the screen driver.
- Editor command keystrokes to be recognised.

These details can be entered manually or loaded from a file. The program has a 'book' of parameters for common screen types, which the user may add to.

The following facilities are also provided:

- Saving the installation details into a file in a form that can be reloaded if required.
- Loading the installation details from a file.
- Output of the installation details to a printer in a readable form.
- Testing the installation details for internal consistency.
- Trying the keystrokes and the screen commands.

4. Running Install

Before running Install, it is a good safety precaution to make a back-up copy of the disc on which Mallard BASIC was supplied. Your operating system user guide should explain how to do this. Then proceed to install one copy, keeping the other safe in case of disaster!

Load your operating system as usual, then replace the disc in Drive A with your BASIC disc. Remember to let the operating system know you have changed the disc if your operating system requires this.

Now enter BASIC as normal by typing:

MALLARD

Appendix VI: Installation of BASIC

After the Ok prompt type:

```
DIR MALLARD.*
```

The screen will then show a filename, probably MALLARD.COM or MALLARD.CMD. This is the filename of the main BASIC interpreter. Remember it, as the installation process will need it.

The Ok prompt will be displayed. Now type:

```
RUN "INSTALL"
```

to run the installation program. During this program you will be prompted for information. Always type the information followed by RETURN. First, a copyright message will appear, followed by the question:

```
Which file do you wish to install?
```

Type the filename of the BASIC interpreter that you displayed just now. You will then get a message asking you if this is correct. Type Y if it is and N if not.

The copyright messages and identification of the BASIC to be installed will then be displayed. Check that this is what you expected and confirm that it is correct.

The installation program's main menu is the displayed.

```
1 start again
2 select standard terminal
3 customise your teminal
4 print installation details
5 space compression ON/OFF          ON
6 editor ON/OFF                     OFF
I install changes into MALLARD.xxx, then exit to system
X exit to system without installing the changes
```

To use the screen based line editor, first choose option 6 by typing 6 and type ON to turn on the editor you are going to install. You should also decide at this stage if you want space compression on or off. We recommend ON for program development.

Now install the editor by giving details of you terminal. These details are already set up for a number of standard terminal types. Select option 2 to list the possibilities in the form of a menu. (Note that the terminal type menu may be more than one screenful: type M to see the next screen.) If your computer is not listed, you can either try selecting one you believe to be similar and see if it works, or you can leave the terminal type menu (by typing X) and select option 3 to customise BASIC for your terminal.

The menu that appears if you opt to customise BASIC contains two choices – customise keyboard and customise VDU. These offer you further menus of features you must set, using the information supplied in the manual for your computer. In particular, the customise keyboard menu asks which keys will be used to perform the various editing functions. The easiest way to answer these questions is just to press the appropriate key, followed by `[RETURN]`. If you are in any doubt, both menus have a help option which explains the details in full.

Having chosen a standard terminal or customised BASIC to your computer, simply type X to leave menus until you return to the main menu. Then choose option I to install BASIC on the disc.

If you decide against installing any changes, type X.

If later you have any problems, or discover that the installation isn't quite right, you can always run `INSTALL` again and make the necessary corrections.

The Command Editor

The command editor is available in all versions of Mallard BASIC except Run Only versions.

It is machine independent and does not require any installation. To achieve this it firstly does not attempt to show the complete up to date state of the line being edited, and secondly uses the minimum of control codes as commands.

The command editor is always in one of three modes – Command mode, Insert mode or Overstrike mode:

Command mode All keystrokes are interpreted as editor commands.

Insert mode Any legal characters typed are inserted into the line at the current line position and displayed at the current cursor position. Any characters to the right of the current line position are moved to the right, but this is not shown on the screen. If the line is full any attempt to add a character will cause a bleep, and the character will be discarded.

A restricted number of control codes are accepted and acted on.

Overstrike mode Any legal characters typed are entered into the line replacing the character at the current line position and displayed at the current cursor position. If the current line position is at the end of the line characters are inserted as in Insert mode.

A restricted number of control codes are accepted and acted on.

When the command editor is invoked the initial contents, if any, of the line to be edited are displayed and the cursor placed at the start of the following screen line. If the line to be edited is not completely empty a prompt is displayed. The editor then starts in Insert mode if the line to be edited is empty, or empty apart from a line number, otherwise it starts in Command mode. The editor may be invoked:

- *by an EDIT command or a syntax error during AUTO when an existing line is found*
by Control-A in Direct Mode when the previous line was a program line

The line number is displayed as a prompt. The editor starts in Command mode.

- *during AUTO where the line does not exist by Control-A when the previous line contained only a line number*

The line number is displayed as a prompt. The editor starts in Insert mode.

- *by Control-A when the previous line was not empty but did not start with a line number*

The string '?' is displayed as a prompt. The editor starts in Command mode.

- *in Direct mode with an empty line*

No prompt is given. The editor starts in Insert mode.

WARNING

Both the move left and the backwards delete commands attempt to keep the display in step with the current line by sending backspaces and spaces to the console. This may fail if the edited line is displayed on more than one console line, or if backslashes and edited characters have been displayed by the 'D', 'H' or 'K' commands. In this case the console does not show the current state of the line so backspacing over it may be misleading. The user is advised to use the 'L' command to re-display the line and position the cursor at the start, after the line number (if any).

1. Control Code Commands Obeyed in all Modes

The following control codes are treated as commands in all modes:

Carriage Return (&H0D)	Finish editing. The rest of the line after the current position is displayed. The line is then acted upon.
Escape (&H1B)	Enter Command mode.
Control-C (&H03)	Abandon editing. The Break message is displayed. The line is not acted upon, but the current contents are not discarded (so may be re- edited if Control-A is then entered).

2. Commands Only Available in Command Mode

Space (&H20)	<p>Move forward one character in the current line.</p> <p>If the current position is at the end of the line then the position is not changed and a Bell (&H07) is sent to the console. Otherwise, the editor sends the current character to the console and moves to the next character in the current line.</p>
Rubout (&H7F) or Backspace (&H08)	<p>Move back one character in the current line.</p> <p>If the current position is at the start of the line then the position is not changed and a Bell (&H07) is sent to the console. Otherwise, the editor sends a Backspace (&H08) to the console and moves to the previous character in the current line.</p>
B or b	<p>Backwards delete.</p> <p>If the current position is at the start of the line then the position is not changed and a Bell (&H07) is sent to the console. Otherwise, the editor sends Backspace (&H08), Space (&H20), Backspace (&H08) to the console and removes the previous character in the current line.</p>
D or d	<p>Forwards delete.</p> <p>If the current position is at the end of the line then the position is not changed and a Bell (&H07) is sent to the console. Otherwise, the editor sends the current character to the console enclosed in backslash characters and removes it from the current line.</p>
H or h	<p>Delete to the end of the line and enter Insert mode.</p> <p>All characters deleted are displayed enclosed in backslashes.</p>
I or i	<p>Enter Insert mode.</p>
K or k	<p>Delete to character.</p> <p>This command deletes all characters up to, but not including, the first occurrence of the character typed following the 'K' or 'k'. The character at the current position is not included in the search, so it is always deleted. All characters deleted are displayed enclosed in backslashes.</p>

L or l	List the current line. Restarts the editor with the current line. This is useful because restarting the editor will cause the current line to be displayed.
O or o	Enter Overstrike mode.
S or s	Search for character. This command moves forward to the first occurrence of the character typed following the 'S' or 's'. The character at the current position is not included in the search, so is always moved past. All characters moved past are sent to the console.
X or x	Extend the current line. The current position is moved to the end of the line and the editor enters Insert mode. All characters moved past are sent to the console.

3. Legal Characters in Insert and Overstrike Modes

The following characters are accepted as text and may be entered into a line in either Insert or Overstrike mode:

- Tab (&H09) Tabs are displayed as spaces to the next tab position
- Line Feed (&H0A) Line Feeds are displayed as Carriage Return (&H0D) followed by Line Feed (&H0A).
- ASCII printing characters in the range &H20...&H7E.
- Additional characters in the range &H80...&HFF.

As well as the control codes given in Section 4.3.1 above, Backspace (&H08) and Rubout (&H7F) are obeyed in Insert and Overstrike modes as follows:

- Insert mode Backwards delete.
If the current position is at the start of the line then the position is not changed and a Bell (&H07) is sent to the console. Otherwise, the editor sends Backspace (&H08), Space (&H20), Backspace (&H08) to the console and removes to the previous character in the current line.
- Overstrike mode Move back one character in the current line.
If the current position is at the start of the line, the position is not changed and a Bell (&H07) is sent to the console. Otherwise, the editor sends a Backspace (&H08) to the console and moves to the previous character in the line.

Any attempt to enter a control character (in the range &H00...&H1F) other than one of those mentioned above (including in Section 1) will be rejected, the character will be ignored and a Bell (&H07) sent to the console.

4. Assumptions Made About the Console

The command editor sends a small number of control codes to the console. These codes are assumed to have effect as follows:

- sending Carriage Return, Line Feed (&H0D, &H0A) is expected to move the cursor to the start of the next console line.
- sending Backspace (&H08) is expected to move the cursor back one character position, without deleting the character. If the cursor is already at the lefthand edge of the console then either the cursor should not move, or it should move to the end of the previous console line.
- sending a Bell (&H07) is expected to produce some sort of warning noise.

These are the generally accepted meanings of these control codes.

Appendix VIII

ASCII Character Set

The following table shows the US ASCII character set assumed in this manual.

While most computers use an ASCII character set, there are often subtle variations in the character used at certain points in the table. In particular, BASIC assumes that the character with internal value &H23 is #, whereas it will be £ if the UK ASCII character set is being used. Other national language variants have further differences and output on your printer can also be affected. The characters which you can expect to be affected are shown shaded below. Of these, #, \ and ^ have special meaning to BASIC (described elsewhere in this manual). When including these characters in BASIC commands, substitute the corresponding character from your computer's character set.

Dec									
	0	16	32	48	64	80	96	112	
Hex	0	1	2	3	4	5	6	7	
0	0	NULL	DLE	Blank	0	@	P	·	p
1	1	SOH	DC1	!	1	A	Q	a	q
2	2	STX	DC2	"	2	B	R	b	r
3	3	ETX	DC3	#	3	C	S	c	s
4	4	EOT	DC4	\$	4	D	T	d	t
5	5	ENQ	NAK	%	5	E	U	e	u
6	6	ACK	SYN	&	6	F	V	f	v
7	7	BEL	ETB	'	7	G	W	g	w
8	8	BS	CAN	(8	H	X	h	x
9	9	HT	EM)	9	I	Y	i	y
10	A	LF	SUB	*	:	J	Z	j	z
11	B	VT	ESC	+	;	K	[k	{
12	C	FF	FS	,	<	L	\	l	
13	D	CR	GS	-	=	M]	m	}
14	E	SO	RS	.	>	N	^	n	~
15	F	SI	US	/	?	O	_	o	DEL

Trigonometrical functions

A range of trigonometrical functions can be derived from BASIC's built-in functions. Expressions for the commonest of these are given below.

Important: These formulae should be used with care, because in a number of cases, there are values of x for which the formula is not valid. These are noted in the table (machine precision value of π).

Function	Equivalent formula	Invalid values of x
SEC(x)	$1/\text{COS}(x)$	$\pi/2, 3\pi/2$ etc.
COSEC(x)	$1/\text{SIN}(x)$	$0, \pi, 2\pi$ etc.
COT(x)	$1/\text{TAN}(x)$	$0, \pi, 2\pi$ etc.
SINH(x)	$(\text{EXP}(x) - \text{EXP}(-x))/2$	
COSH(x)	$(\text{EXP}(x) + \text{EXP}(-x))/2$	
TANH(x)	$(\text{EXP}(x) - \text{EXP}(-x))/(\text{EXP}(x) + \text{EXP}(-x))$	
SECH(x)	$2/(\text{EXP}(x) + \text{EXP}(-x))$	
COSECH(x)	$2/(\text{EXP}(x) - \text{EXP}(-x))$	$x = 0$
COTH(x)	$\text{EXP}(x)/(\text{EXP}(x) - \text{EXP}(-x))$	$x = 0$
$\text{SIN}^{-1}(x)$	$\text{ATN}(x/\text{SQR}(-x^2+1))$	$x = \pm 1$
$\text{COS}^{-1}(x)$	$-\text{ATN}(x/\text{SQR}(-x^2+1)) + \pi/2$	$x = \pm 1$
$\text{SEC}^{-1}(x)$	$\text{ATN}(x/\text{SQR}(x^2-1)) + \text{SGN}(\text{SGN}(x)-1)*\pi/2$	$x = \pm 1$
$\text{COSEC}^{-1}(x)$	$\text{ATN}(x/\text{SQR}(x^2-1)) + (\text{SGN}(x)-1)*\pi/2$	$x = \pm 1$
$\text{COT}^{-1}(x)$	$\text{ATN}(x) + \pi/2$	
$\text{SINH}^{-1}(x)$	$\text{LOG}(x + \text{SQR}(x^2+1))$	
$\text{COSH}^{-1}(x)$	$\text{LOG}(x + \text{SQR}(x^2-1))$	
$\text{TANH}^{-1}(x)$	$\text{LOG}((1+x)/(1-x))/2$	$x = 1$
$\text{SECH}^{-1}(x)$	$\text{LOG}((\text{SQR}(-x^2+1)+1)/x)$	$x = 0$
$\text{COSECH}^{-1}(x)$	$\text{LOG}((\text{SGN}(x)*\text{SQR}(x^2+1)+1)/x)$	$x = 0$
$\text{COTH}^{-1}(x)$	$\text{LOG}((x+1)/(x-1))/2$	$x = 1$

Binary Floating Point Arithmetic

Mallard BASIC uses Binary Floating Point Arithmetic when dealing with values with fractional parts. This Appendix describes some of the properties of floating point arithmetic in general, and binary floating point in particular. Some effects of binary floating point surprise some users, and may need to be taken into account even in simple programs. Other effects will only be of interest in numerically intensive programs.

All floating point arithmetic systems have some limit to the precision to which values are held and new values generated. For many purposes the error involved causes no ill effects. A badly constructed program can, however, result in dangerous magnification of the intrinsic errors. There is a complete branch of mathematics – Numerical Analysis – concerned with the construction of accurate numerical algorithms. This appendix can only touch on some of the issues involved. If in doubt, the programmer should consult an expert.

This Appendix has a number of sections, as follows :

1. Floating Point Arithmetic, a Brief Introduction.

This section describes how floating point values are stored and operated on. The source and effect of the (small) errors implicit in floating point arithmetic are examined. This is offered as 'background material', to support the more specific recommendations given in later sections.

2. Decimal Fractions and Binary Floating Point.

This section describes the effects of using binary floating point numbers to represent decimal fractions. Mallard BASIC uses this technique, and the effects affect most users.

3. Binary Floating Point and Financial Calculations.

This section contains specific recommendations for programs performing calculations on money values.

4. Binary Floating Point and Scientific/Engineering Calculations.

This section contains specific recommendations for programs performing calculations in scientific or engineering applications.

1. Floating Point Arithmetic, a Brief Introduction

The analysis of the source and effects of errors in sequences of floating point operations is a non-trivial task. If in doubt the programmer's simplest option is to use double length arithmetic at all times. Note that this really only postpones the problems – though, with the exception of the subtraction of very similar values, this postponement will usually be indefinite.

Floating point arithmetic is a very powerful tool, but like many such tools it should be used with care. The examples given are, perhaps, a little extreme. They are intended to illustrate the potential problems, they are not intended to alarm.

Floating Point Number Representation

Floating point numbers have two parts, a 'significand' (or 'mantissa') and an 'exponent'. The significand is usually either a fraction, or a units digit plus a fraction. The exponent gives the power of the 'radix' to multiply the significand by, to give the actual value. In a decimal floating point system (ie. one where the radix is 10) a value is represented by:

$$\text{significand} \times 10^{\text{exponent}}$$

The significand will have a restricted number of digits, which dictates the precision to which any number can be held. The size of the exponent will also be limited, which limits the range of values which can be represented.

For example, consider a decimal system with six digit significands, in which the significand is units plus fraction. Some values, chosen more or less at random, and their floating point representations are:

299,792,512.0537	2.99793×10^8
3.1415926535897932	3.14159×10^0
0.0000453999297	4.53999×10^{-5}

The exponent value is chosen such that the significand is always at least 1, but less than 10. This means that no matter how big or small the number, the number of significant digits (the precision) is always the same – it is this property of floating point numbers which makes them so useful. Note also that the floating point representation is not exact, but it is rounded to the nearest value possible in the digits available.

Binary floating point systems are no different, except that the radix is 2 and the significand is a binary fraction or one plus a binary fraction. In Mallard the significands are fractions greater than or equal to a 2 and less than 1. The single and double precision numbers have significands of 24 binary digits and 56 binary digits, respectively, giving approximately 7 and 16 significant decimal digits.

The floating point representation of a given number x may be written:

$$x \times (1 + \epsilon)$$

where ϵ is the relative error introduced when rounding the true value to the available precision, and $x \times \epsilon$ is the actual error. In Mallard the maximum relative error in single precision is $\pm 2^{-24}$ (approximately $\pm 6.0 \times 10^{-8}$), in double precision the maximum relative error is $\pm 2^{-56}$ (approximately $\pm 1.4 \times 10^{-17}$). [The relative error is (actual value – true value) divided by the true value.]

Floating Point Multiplication and Division

In multiplication the significands are multiplied together and the exponents are added. The multiplication produces twice as many significant digits as in each argument. In division the significands are divided and the exponents subtracted. The division may produce an indefinite number of significant digits. In both cases it is necessary to round the result to the number of digits available for the significand.

Given values x and y to be multiplied together the operation is:

$$(x \times (1 + \epsilon_1)) \times (y \times (1 + \epsilon_2))$$

where ϵ_1 and ϵ_2 are the relative errors introduced by rounding the true values to the floating point representations. The result is :

$$(x \times y \times (1 + \epsilon_1 + \epsilon_2 + \epsilon_1 \times \epsilon_2)) \times (1 + \epsilon_3)$$

where ϵ_3 is the relative error introduced by the rounding of the result. Since the relative errors are very small, second and third order errors (any term involving an error multiplied by another) may be ignored, giving:

$$x \times y \times (1 + \epsilon_1 + \epsilon_2 + \epsilon_3)$$

The three errors are of similar size, so the result may contain an error three times as large as the original errors. Some of the errors may be negative, so they may combine favourably. Proper error analysis must, however, assume worst case errors. In Mallard the maximum relative error in single precision is $\pm 2^{-25}$ so three times that is still not a large error. In double precision the maximum relative error is $\pm 2^{-57}$ so even several multiples of it is still vanishingly small.

Similar analysis can be applied to division.

The operation on the exponents can yield a value which is beyond the range available. If the resulting exponent is greater than the largest positive exponent possible, then the number is simply too big to be represented – this is called Overflow. Generally the floating point system will have a value to represent all numbers which are too large to represent, which is often known as ‘machine infinity’. In the event of overflow Mallard produces an error message and returns the largest representable number. If the resulting exponent is less than the most negative exponent possible, then the number is too small to be represented – this is called Underflow. Mallard does not take any special steps to deal with underflow, the result is simply treated as zero. This means that the sign of the result is lost, since zero has no sign, and that a future division may report a divide by zero error.

Floating Point Addition and Subtraction.

These operations are more complicated than multiplication and division. To add or subtract two floating point values it is necessary to divide the significand of the smaller number to make the exponents the same. Using the example decimal system the following addition :

$$3.14159 \times 10^0 + 4.53999 \times 10^{-5}$$

requires the second significand to be divided so that the sum is :

$$3.14159 \times 10^0 + 0.0000453999 \times 10^0$$

giving 3.141635399×10^0 , which when rounded to six digits is 3.14164×10^0 .

The problem with addition and subtraction is that when the numbers do not have similar exponents (ie when they are not of a similar size) some significance is lost. In the example above the result of the operation would have been the same if the smaller value had been 5×10^{-5} , so most of the digits in the smaller number have simply been ignored! The actual error introduced in this case is $.46001 \times 10^{-5}$ which is a relative error of approximately 1.5×10^{-6} . This is not a serious problem for a single addition, but can badly affect multiple additions, so that, for example:

$$3.14159 \times 10^0$$

$$+ 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5} \\ + 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5} + 4.53999 \times 10^{-5}$$

gives 3.14209×10^0 , while :

$$3.14159 \times 10^0 + (4.53999 \times 10^{-5} \times 10)$$

gives 3.14204×10^0 , which is closer to the true value, and visibly different to the result of multiple additions.

Furthermore, when adding or subtracting values of quite different size the error in the representation of the larger may become more significant. Thus, adding two values x and y gives:

$$(x \times (1+\epsilon_x)) + (y \times (1+\epsilon_y))$$

which is :

$$(x+y + x\epsilon_x + y\epsilon_y) \times (1+\epsilon_r)$$

where the $(1+\epsilon_r)$ term reflects the error in rounding the result. The relative errors are all of similar size. If y is small compared to x , then the term $x\epsilon_x$ becomes substantial compared to y , with an unfortunate effect on the significance of the result. In the example above the actual error in the representation of 3.1415926535897932 is $-0.26535897932 \times 10^{-5}$, which is similar in size to the 4.53999×10^{-5} which was being added!

Floating point subtraction of values of similar size can produce still more unpleasant effects. Consider two positive numbers x and y , of similar size. Subtracting the floating point versions can be written as:

$$(x \times (1+\epsilon_x)) - (y \times (1+\epsilon_y))$$

and the result is :

$$(x-y + x\epsilon_x - y\epsilon_y) \times (1+\epsilon_r)$$

where the $(1+\epsilon_r)$ term reflects the error in rounding the result. Now $x-y$ is small (because x and y are of similar size) so that despite the fact that ϵ_x and ϵ_y are small $x\epsilon_x$ and $y\epsilon_y$ may be large compared to the true result. Using the same decimal system used above, the following illustrates this point:

$$3.1415926535897932 - 3.1414379573205291 = 1.546962692641 \times 10^{-4}$$

becomes :

$$3.14159 \times 10^0 - 3.14144 \times 10^0 = 1.50000 \times 10^{-4}$$

which is a relative error of in excess of 3×10^{-2} !

2. Decimal Fractions and Binary Floating Point

As pointed out in various parts of this manual, many (indeed most) decimal fractions simply do not have exact representations in binary. The deceptively simple decimal fraction .1, for example, is the recurring binary fraction .00011001100110011...! When such a decimal fraction is converted to binary it is rounded to the nearest representable value. While the result is precise to within the limits given for the form of number, it is not exactly the original fraction.

Appendix X: Binary Floating Point Arithmetic

When fractional values are printed by Mallard the binary numbers are converted to decimal, which involves rounding to either 7 or 16 significant decimal digits. The inexactness of the representation may, therefore, be hidden. Furthermore, it is possible that values which are actually different will produce the same result when printed – the difference may only come to light if the values are compared or subtracted. It is quite easy to produce examples where the original inexact representation is compounded by arithmetic operations. Thus:

```
PRINT 5 - (0.49 * 10)
```

produces : 0.0999999

which is very close to the true result of 0.1, but is not precisely the same ! Note that this example demonstrates two effects. The first is that the binary floating point representation of 0.49 is very slightly larger than its true value. The second is that the subtraction of two values of similar size tends to amplify any existing errors. If the same example is converted to double length then:

```
PRINT 5# - (0.49# * 10)
```

produces : 9.999999999999998D-02

which is even closer to the true result, but not quite there!

It is important to understand that this and related effects are a direct consequence of the use of binary floating point. This is not a fault in Mallard BASIC's arithmetic, all binary floating point systems work this way. It is possible to perform decimal floating point arithmetic on computers. Mallard BASIC uses binary floating point simply because the computers used are more suited to it, and therefore the binary floating point operations are substantially faster than their decimal equivalents.

The ROUND function is provided to round a value to a given number of decimal places. So the examples above can be modified thus:

```
PRINT ROUND(5 - (0.49 * 10), 6) produces : 0.1
```

```
PRINT ROUND(5# - (0.49# * 10), 16) produces : 0.1
```

Note that rounding to a particular number of decimal places is not necessarily the same as rounding to a particular number of significant digits. For example the number 0.00017236 rounded to four decimal places is 0.0002, but rounded to four significant digits is 0.0001724. There is no built-in function to round to a number of significant digits, but the following user defined function may be used:

```
DEF FNs#(x#,n) = VAL(DEC$(x#,"+."+STRING$(n,"#")+"^^^^"))
```

which rounds the value $x\#$ to n significant decimal digits.

For loops should not be used with fractional steps. The command `FOR i=0 TO 100 STEP 0.1` produces i s which are increasingly far from the value intended, with the last value of 99.99905. To get the desired effect something like `FOR ii=0 TO 1000 : i=ii/10` should be used.

3. Binary Floating Point and Financial Calculations

Floating point in general, and binary floating point in particular, are not ideally suited to financial calculations. While scientific and engineering calculations are quite happy with results which are accurate to a given number of significant digits, most financial calculations must be exact. The programmer must, therefore, take steps to ensure that the results of operations on money values are indeed exact. The following are recommended :

a. Use Double Precision Arithmetic.

Mallard BASIC's single precision arithmetic provides some 7 significant decimal digits. This is fine for relatively small sums, but £10,000.00p already contains 7 significant digits – so calculations involving these amounts of money are already at the limits of single precision. Meaningful quantities of Italian Lire definitely require double precision.

The DEFDBL command may be used to set all variables which do not have an explicit type to be double precision (the default is for these to be single precision). Constants in a program may be declared to be double precision by appending the '#' type marker. This is most important, because there is a real difference between 0.1 and 0.1#. The first is a single precision value, the second is a double precision value, thus :

```
a# = 0.1
b# = 0.1#
PRINT a# - b#
gives: 1.490116119037821D-09
```

which reflects the difference in precision between the two representations.

b. Use ROUND.

Where the result of a calculation is a sum of money, then ROUND should be used to round to the required number of decimal places (typically 2). For example :

```
a# = 90.92# +87.14# +15.84# +21.25# +74.61#
b# = 289.76#
PRINT a#, b# - a#, b# - ROUND(a#, 2)
gives: 289.76 7.105427357601002D-15 0
```

which demonstrates what can happen when adding a column of figures together and reconciling it with a figure, which has been calculated in a different way. The difference is minute, but the extra ROUND step ensures that it is actually zero.

c. Beware of Rounding

If the five values used above are treated as prices, which are to be discounted by 45% then adding together the rounded discounted prices gives the result 159.38. Unfortunately applying the same discount to the rounded total of the original prices gives 159.37! This effect is nothing to do with the floating point arithmetic. Both results are correct, reflecting two different approaches to the problem of discounting multiple items.

4. Binary Floating Point and Scientific/Engineering Calculations

Floating point is well suited to these sorts of problems. Where results are required to four or five significant decimal digits single precision arithmetic should be adequate. Where greater accuracy is required double precision is recommended, though very careful use of single precision can provide up to seven significant decimal digits. (Note, however, that the built-in arithmetic functions force their arguments to single precision and produce single precision results.)

The analysis of errors arising during floating point arithmetic is touched on in the Brief Introduction to Floating Point Arithmetic above. The following recommendations do not obviate the need for the careful design of complex numerical algorithms, but should help with simple problems :

a. Do not test for equality.

It is very unlikely that the statement(s) after the THEN in `IF x = 7.41 THEN ...` will ever be executed – unless the value 7.41 is assigned directly to x! The alternatives are to round x so that the test takes into account the acceptable margins of error for the problem, or to test that x lies in some range suitable to the problem.

b. Beware of subtractions of values of similar size.

As explained above these can produce quite badly flawed results, because any errors in the operands can be hugely magnified. It may be necessary to examine the result of a subtraction, and treat it as zero if its absolute value is less than some minimum suitable to the problem (ROUND may be used to do this, as well).

c. Beware of repeated additions.

Because each addition can contribute a rounding error to the result it is best to either use a single addition of a value which is multiplied up, or use double precision for the repeated operations.

d. Beware of INT and FIX.

These functions produce integers from floating point values. They are very literal minded, for example `INT(2.1 - 0.1)` gives 1, as does `INT(2.1# - 0.1#)` ! This is because the errors in the representation of 2.1 and 0.1 conspire to make the result of the subtraction just short of 2, so INT truncates the value to 1. (Note that simply printing the result of the subtraction produces the value 2, because there is further rounding in the binary to decimal conversion.) The FIX function is equally brutal. The solution, as ever, lies in examining the levels of accuracy required by the problem and applying them – for example `INT(ROUND(x, 5))` may be suitable for some applications.

e. Beware of addition and subtraction of values of widely differing sizes.

Adding a small number to a much larger one can lead to loss of significance. This can be demonstrated by `(0.2 + 100000) - 100000` which gives 0.203125, which represents a large relative error. These effects may be mitigated by using double precision.

Index

- Abandon editing, 150
- Abandoning a program, **44**
- ABS, **54**, 199
- Absolute value, 199
- Addition, **5**, 136
- ADDKEY, **106**, 200
- ADDREC, **97**, 202
- Address
 - highest used, 389
- Address of variables, 379
- address-expression*, 195
- Allocating memory
 - for keyed access files, **93**, 206
 - machine code programs, 117
- Amending programs, 244
- AND, **43**, **55**, 139
- Arc-tangent, 204
- Arithmetic, **5**, **53**, 153
- Arrays, **24**
 - dimensions, **24**, 241
 - elements, **24**
 - erase, 248
 - initialisation, **24**, 338
 - lower bound, 133, 315
 - subscripts, 133
 - variable, 133
- array-variable-name*, 195
- ASC, **59**, 203
- ASCII character set, **23**, 421
- Assignment, 281
- ATN, **54**, 204
- AUTO, **11**, 145, 205
- Based numbers, **21**, 130, 132
- BASIC
 - leaving, **1**
 - starting, **1**
- Bitwise operations, **55**
- Branching, **41**
- Break, 142, 146
 - see also Control-C
 - resume after, 222
- BUFFERS, **93**, 206
- Built-in functions, 139
- CALL, **119**, 208, 400
- Carriage return
 - in sequential files, 177
- CD, 175, 209
- CDBL, **60**, 210
- CHAIN, 211
- CHAIN MERGE, 211
- Character set, 129, 421
- Characters
 - ASCII values, 203
 - convert to integer, **59**, 203
 - input from keyboard, **19**
 - internal value, **23**
 - literal, 148
- CHDIR, 175, 213
- CHDIR\$, 175, 214
- CHR\$, **59**, 215
- CINT, **60**, 216
- CLEAR, 187, 217
- Clear variables except common, 220
- CLOSE, 68, 177, 218
- Codes
 - escape sequences, **29**
 - for printer control, **34**
 - for screen control, **29**
- Comma
 - in print command, 164

- Command Editor, 415
- Command line parameters, 389
- Command structure, 2
- Commands
 - console I/O, 154
 - constant data, 158
 - control structures, 153
 - conversion to upper case, 129
 - creation of programs, 151
 - definition of terms, 195
 - directory access, 156
 - error trapping, 160
 - file input and output, 155
 - keyed files, 188
 - machine level, 160
 - output to printer, 155
 - overview, 151
 - program development, 161
 - program termination, 152
 - random access files, 157
 - running programs, 152
 - sequential files, 156
 - structure, 127
 - syntax, 195
 - variables, 154
- Comments, 37, 339
- COMMON, 219
- COMMON RESET, 220
- Computed GOSUB, 311
- Computed GOTO, 311
- Concatenating strings, 56, 137
- Conditionals, 41, 269
- Consistent files, 221
- Console
 - input fixed length string, 275
 - input line, 282
 - output data, 386
 - position, 331
 - trap input, 318
 - trap output, 323
 - width of, 384
- Console I/O, 154
- CONSOLIDATE, 183, 221
- Constants, 3, 16, 231, 338, 343
- CONT, 44, 222
- Continuation records, 79
- Control codes, 23, 29, 169
 - interpretation, 170
 - specification, 170
- Control variable, 261
- Control-A, 127, 145
- Control-C, 127, 142, 146, 153
 - disable, 324
 - enable, 325
 - resume after, 222
- Control-S, 127, 142, 153
 - disable, 324
 - enable, 325
- Conversion functions, 59
- COS, 54, 223
- Cosine, 223
- CREATE, 224
- Creation of programs, 151
- CSNG, 60, 225
- Current directory
 - changing, 209, 213
 - returning current directory, 214
- Current position, 186
- Cursor control, 29
- CVD, 83, 180, 226
- CVI, 83, 180, 227
- CVIK, 228
- CVS, 83, 180, 229
- CVUK, 230
- DATA, 16, 231
 - disc files, 20, 61
 - disc storage, 61
 - entry from the keyboard, 17
- Data base programs, 89
 - writing, 91
- Data files, 90, 181

- Data formats, 397
- Data records, 76
- Data types, 131
 - conversion, 134
- Debugging, 373
- DECS, 232
- Decimal fractions, 429
- Decimals, 21
- DEF FN, **50**, 139, 233
- DEF SEG, 234
- DEF USR, **120**, 235
- Defaults, 389
 - disc drive, 317
 - user number, 317
 - variable type, 132, 236
- DEFDBL, 236
- DEFINT, 236
- DEFSNG, 236
- DEFSTR, 236
- DEL, **62**, 175, 237
- DELETE, **13**, 238
- Delete files, 237, 247, 278
- DELETE option, 211
- Deleting program lines, **13**, 238
- DELKEY, **102**, **107**, 239
- DIM, **25**, 241
- Dimensions of an array, **24**, 133, 241
- DIR, **62**, 174, 242
- Direct mode, 127, 141
 - special run only, 142
- Directory, 242, 258
 - changing current directory,
209, 213
 - create new directory, 296, 302
 - directory lookup, 259
 - listing, 257
 - remove directory, 337, 347
 - returning current directory, 214
 - tree-structured, 175
- Directory access, 156
- Directory operations, 174
- Disc commands, **62**, 174
- Discs
 - changing, 342
- DISPLAY, 243
- Displaying programs, **11**
- Division, **5**
 - by zero, 137
- Double Length values, 131, 174, 210
 - exact representation, 140
 - in random records, 226, 301
 - output, 165
- Double precision numbers, **22**
- drive-spec*, 198
- Duplicate keys, 336
- EDIT, 145, 244
- Editing programs, **12**
 - keystrokes, **13**
- Editor, 141, 145
 - installing, 409
 - leaving, 149
- Elements of an array, **24**
- ELSE, **42**, 269
- END, **44**, 245
 - resume after, 222
- End of file, **67**, 246
- End of line, 177
- EOF, **67**, 246
 - in sequential files, 178
- EQV, 139
- ERA, **62**, 175, 247
- ERASE, 248
- Erasing files, 175, 237, 247, 278
- ERL, 249
- ERR, 250
- ERROR, 251
- Error processing mode, 142, 312, 344

Errors

- error messages, 391
- error numbers, 250, 391
- force from program, 251
- from operating system, 327
- keyed files, 192
- error line number, 249
- resume execution, 344
- trapping, 160, 312

ESC, 29

Escape sequences, 29

Exiting from BASIC, 1, 370

EXP, 54, 252

Exponential, 252

Exponentiation, 136

Exponents, 54

- formatted printing, 168

Expressions, 5

- evaluation of, 5
- logical, 138
- numeric, 135
- relational, 55, 137
- string, 137

expression, 195

External functions, 120, 140, 235, 377

- entry and exit conditions, 398

External routines, 208, 397

F: command line parameter, 389

FETCHKEY, 253

FETCHRANK, 254

FETCHREC, 255

FIELD, 78, 179, 256

Field variables, 78

- assigning information, 79

File handling

- examples, 69, 84, 112
- overview, 173

File lock, 176

File names, 174

- file name lookup, 258

file-name, 195

file-name-expression, 195

File numbers, 176

file-number-expression, 195

file-reference, 195

FILES, 174, 257

Files

- closing, 176, 218

- current record number, 286

- deleting, 62, 237

- directory, 242, 257, 258

- displaying contents, 62, 243

- erasing, 175, 247, 278

- indexed, 89

- input data, 274

- input fixed length string, 275

- input line, 283

- input random record, 264

- keyed access, 89, 181

- length of file, 288

- listing, 62

- locking, 183

- maximum number open

 - simultaneously, 63, 217, 297, 389

- opening, 176, 313

- output data, 333, 387

- output to console, 243, 374

- random access, 76, 178

- records, 76

- renaming, 62, 175, 307, 340

- sequential, 63, 177

FIND\$, 62, 174, 258

FINDDIR\$, 175, 259

Finish using BASIC, 1

FIX, 60, 260

Floating point numbers, 22, 131

- convert to integer, 60

- exact representation of, 140

Floating point arithmetic

- inaccuracy, 26, 425

- warnings, 425

- Flow of control, **38**
- FN, **233**
- FOR, **39**, **261**
- Formatted printing, **166**
- Free format printing, **165**
- Format template, **28**
- FRE function, **263**
- Free space, **263**
- Functions, **6**, **50**, **139**, **377**
 - arithmetic, **6**, **158**
 - call, **50**
 - define external function, **235**
 - external functions, **120**, **398**
 - machine level, **160**
 - overview, **151**
 - parameters, **50**
 - position in program, **50**
 - string, **6**, **159**
 - trigonometric, **54**, **423**
 - type conversion, **159**
 - user defined, **50**, **233**
- Garbage collection, **263**
- Generation number, **183**
- GET, **82**, **179**, **264**
- GOSUB, **47**, **265**
- GOTO, **39**, **47**, **266**
 - in IF command, **269**
- GSX routines, **122**
- Guard record, **407**
- Halt, **142**
- HEX\$ function, **267**
- Hexadecimal numbers, **21**
- Hexadecimal value, **267**
- Highest address used, **389**
- HIMEM, **117**, **268**
- I/O port, **271**, **328**, **381**
- IF, **41**, **269**
 - sequence of options, **42**
- IMP, **139**
- Index files, **90**, **181**
- index-position*, **197**
- Information storage
 - in keyed access files, **97**
 - in random records, **79**
 - on disc, **61**
- Initialising BASIC, **389**
- INKEY\$, **19**, **270**
- INP, **271**
- INPUT, **18**, **272**
- Input
 - characters, **19**
 - from console, **154**, **272**, **275**, **282**
 - from disc files, **20**
 - from I/O port, **271**
 - from keyboard, **17**, **270**
 - from random record, **274**, **275**, **283**
 - from sequential file,
 - 156**, **274**, **275**, **283**
 - numbers, **18**
 - prompted, **272**
 - strings, **19**
- INPUT #, **65**, **274**
 - with random files, **179**
- INPUT \$, **275**
- INPW, **271**
- Installation of BASIC, **409**, **411**
- INSTR, **58**, **276**
- INT, **60**, **277**
- Integer division, **136**
- Integer modulus, **136**
- Integers, **21**, **131**
 - convert to character, **59**
 - convert to floating point, **60**
 - convert to hexadecimal, **267**
 - convert to octal, **310**
 - in random records, **227**, **303**
 - range, **130**
 - use as keys, **228**, **230**, **304**, **306**

integer-expression, 195

Iteration, **44**, 261

Jetsam

see also Keyed access files

red tape, 316

Joining strings, **56**

Jumps (in programs), **39**, 265, 266

Key value, 181

key-value, 197

Keyboard

input characters, 270

input from, 17

installation, 410

Keyed access files, **89**, 405

adding extra keys, **106**, 200

adding records, **95**, 191, 202

allocating memory, **93**

build record, 294, 299, 350

change lock, 287

changing an entry, **108**

changing the key, **107**

closing, **91**, **103**, 189

consistent files, 182, 221

creating, **91**, 188, 224

current key rank, 254

current key value, 253

current position, **90**, 186

current record number, 255

defining records, **94**

deleting a key, 239

deleting a record, **101**, 191, 239

duplicate keys, 336

errors, 192

example, **112**

generation number, 183

input record, 264

integer keys, 228, 230, 304, 306

locks, **90**, 183

maximum record length, 187

multiuser, 407

opening, **94**, 189, 313

output record, 334

overview, 181

rank, **97**, 182

reading a record, **99**, 190

record length, **94**

reserved bytes, 316

restore record, 358

restrictions, 187

return codes, 192

seek record, 354, 355, 356, 359

set memory space, 206

storing information, **97**

writing, 190

Keyed file, 181

Keys, **89**, **97**

adding extra keys, **106**, 191, 200

changing the key, **107**

current rank, 254

current value, 253

deleting, **101**, 191, 239

duplicate keys, 336

finding keys, 191

integer values, 228

numeric values, 405
representation of integer values,
230, 304, 306

sets, **111**

subkeys, 406

Keywords, **2**, 129

summary, 403

KILL, 175, 278

LEFT\$, **56**, 279

LEN, 280

LET, 281

Line feed

in sequential files, 177

- LINE INPUT, 282
 - keystrokes, 146
- LINE INPUT #, 283
- Line numbers, **11**, 205
 - automatic, **11**
- Line-number, 130, 196
- line-number-range*, 196
- LIST, **12**, 284
- List-of*, 128
- Listing programs, **11**, 174
- Literal characters, 148
- LLIST, 284
- LOAD, 285
- Loading programs from disc,
 - 10**, 152, 285
- LOC, 286
- LOCK, 287
- Locks, 176, 197
 - failure, 194
 - file locks, 183
 - record locks, 183
 - temporary, 185
- LOF, 288
- LOG, **54**, 289
- LOG10, **54**, 290
- Logarithms, **54**, 289, 290
- Logical expressions, **41**, 138
- logical-expression*, 138, 196
- Logical operators, 138
- Logical position, 164
- Loops, **38**, 153
 - FOR loops, **39**, 261
 - nesting, 262
 - WHILE loop, **44**, 382, 383
- LOWER\$, 291
- LPOS, 292
- LPRINT, 163, 293
- LSET, **79**, 180, 294
- M: command line parameter, 389
- Machine code programs, **115**
 - call, **119**
 - examples, **120**
 - loading, **118**
 - using, **119**
- Machine level operations, **115**
- MAX, **56**, 295
- Maximum value, 295
- MD, 175, 296
- MEMORY, **117**, 187, 297
- Memory location
 - reading, 329
 - writing, 330
- MERGE, 211, 298
- Merging programs, 211
 - protected programs, 212, 298
- Metalanguage, 128
- MID\$, **56**, **79**, 180, 299
- MIN, **56**, 300
- Minimum value, 300
- MKDS\$, **80**, 180, 301, 405
- MKDIR, 175, 302
- MKI\$, **80**, 180, 303
- MKIK\$, 304, 405
- MKSS\$, **80**, 180, 305, 405
- MKUK\$, 306, 405
- MOD, **53**, 136
- Modes
 - of operating BASIC, 142
- Modulus, 136
- Multiplication, **5**
- Multiuser versions, 181, 183, 405
 - open files, 314
 - random files, 407
 - sequential files, 406

- NAME, 175, 307
- Names, 129
 - of variables, 3, 132
 - path names, 198
 - upper & lower case, 20
 - variable names, 20
- Natural logarithm, 289
- Nesting
 - of loops, 262
 - of subroutines, 265
- NEW, 308
- NEXT, 39, 261, 309
- NOT, 43, 55, 139
- Null characters, 131
- Numbers, 22, 130
 - absolute value of, 54
 - arithmetic, 53
 - based, 21
 - convert to string, 59
 - floating point, 22
 - hexadecimal, 21
 - in random records, 80
 - input from keyboard, 18
 - maximum and minimum values, 56
 - octal, 21
 - precision, 22
 - random, 55
 - scaled, 21
 - sign of, 54
 - type conversion, 134
 - unscaled, 21
 - unsigned integer, 135
 - variables, 3
- Numeric expressions, 135
- Numeric values, 131
 - convert to character, 215
 - convert to double length, 210
 - convert to formatted string, 232
 - convert to hexadecimal, 267
 - convert to integer, 216, 277, 375
 - convert to octal, 310
 - convert to single length, 225
 - convert to string, 366
 - exact representation, 140
 - formatted printing, 167
 - in random access files, 180
 - output, 165
 - rounding, 134, 349
 - sign of, 360
 - truncate, 260
- numeric-constant*, 196
- numeric-expression*, 196
- numeric-variable*, 196
- OCT\$, 310
- Octal numbers, 21
- Octal values, 310
- ON ERROR GOTO, 312
- ON *expression*, 311
- ON GOSUB, 48, 311
- ON GOTO, 42, 311
- OPEN, 63, 176, 313
- Operating system, 380
 - return to, 1
- Operators, 5
 - arithmetic, 53
 - logical, 138
 - on numeric expressions, 136
 - precedence, 5, 136
 - relational, 43, 138
 - string, 138
- OPTION BASE, 315
- OPTION FIELD, 316
- OPTION FILES, 317
- OPTION INPUT, 318
- OPTION LPRINT, 321
- OPTION NOT TAB, 32, 171, 322
- OPTION PRINT, 323
- OPTION RUN, 153, 324
- OPTION STOP, 153, 325

- OPTION TAB, 326
- OR, **43**, **55**, 139
- OSERR, 327
- OUT, 328
- Output, **27**, 163
 - files on console, 374
 - record to random files, 334
 - spaces, 363
 - to console, 154, 332
 - to I/O port, 328
 - to random record, 333, 387
 - to sequential file, 157, 333, 387
 - to the console, 386
 - to the printer, **34**, 155, 293
 - to the screen, **27**
- OUTW, 328
- Overflow, 137
 - in formatted printing, 168
- Overlaying programs, 211
- Parameters
 - command line, 389
- Patches, **115**
- path*, 198
- Path names, 198
- PEEK, 329
- POKE, **116**, 330
- POS, 331
- Precedence
 - logical operators, 138
 - numeric operators, **5**, 136
- Precision of numeric values, 131
- PRINT, **27**, 163, 332
- Print
 - see also Output
 - separators, **27**
 - spaces, 363
- PRINT #, **64**, 163, 333
 - with random files, 179
- Print functions, **28**
- Print items, 164
- PRINT USING, **29**, 332
- Print zones, **28**, 165
 - set size, 388
- Printer
 - output data, 155, 293
 - position, 292
 - trap output, 321
 - width of, 385
- Printer control, **34**, 169
 - codes, **34**
 - Diablo 630 codes, **35**
 - FX-80 codes, **36**
- Printing
 - columns, **29**
 - control character handling, 164
 - exponents, 168
 - field overflow, 168
 - files, 243
 - formatted, **28**, 166
 - logical position, 164
 - numeric values, 165
 - signed numbers, 168
 - spaces, 166
 - template for, **28**
- Procedures, **46**, 265
- Program lines
 - delete, 238
- Program mode, 127, 141, 142
- Programming
 - design, **15**, **37**
 - top down, **48**
- Programs
 - ASCII format, 353
 - automatic numbering, 205
 - chaining, 211
 - changing, **12**, 141, 145, 244
 - comments, **37**
 - conditional flow of control, 269
 - continuing after stopping, **44**, 222
 - decisions, **40**

- deleting from program lines, 149
- deleting program lines,
 - 13, 141, 145
- designing, 15
- designing using modules, 48
- displaying, 11, 284
- editing, 12, 141, 145, 244
- ending, 245
- first steps, 9
- flow of control, 38
- input, 141
- jumps, 39
- line numbers, 11
- loading from disc, 10, 152, 285
- loops, 39
- machine code, 115
- merging, 298
- output to printer, 284
- overlaying, 211
- prepare for new program, 308
- protected, 353
- renumbering, 13, 341
- running, 10, 152, 351, 352
- save on disc, 353
- screen lines, 147
- stopping, 44, 245
- storing on disc, 10
- structure, 37, 45
- termination, 152
- writing a data base program, 91
- writing programs, 37
- Prompt string, 19
- Protected programs, 353
- Pseudo random numbers, 55
- PUT, 81, 179, 334
- Quote marks, 23
- quoted-string*, 196
- Random access files, 76, 178
 - build record, 294, 299, 350
 - change lock, 287
 - changing, 84
 - closing, 81, 83
 - creating, 77
 - data from record, 279, 346
 - defining the record format, 179, 256
 - double length values, 226, 301
 - examples, 84
 - input record, 264
 - layout of data, 78
 - multiuser, 407
 - opening, 77, 82, 313
 - output record, 334
 - reading, 81, 82
 - records, 76
 - writing records to disc, 81
- Random numbers, 55, 348
 - set seed, 335
- RANDOMIZE, 335
- Rank, 97, 182, 197, 357
- RANKSPEC, 336
- RD, 175, 337
- READ, 16, 338
 - see also Input
- Record buffer, 78, 178
- Record length
 - initialise, 217
 - keyed access files, 94
 - random access files, 78
 - set in OPEN, 313
- Record locking, 183, 206
- record-number*, 196
- Record number, 77, 181
 - current record, 255
- Record size, 178

Records, 76

- add to keyed file, 202
- changing the data in, 299, 350
- continuation records, 79
- defining, **78**, 256
- delete keyed, 239
- find by key, 186
- input random record, 264
- maximum record length, 297
- putting information in, 79
- seek by key, 354
- seek different key, 359
- seek next, 355
- seek previous, 356
- writing to disc, **81**

Relational operators, 55

- relational-operator*, 137
- relational-expression*, 137

REM, 37, 339

Remarks, 37

REN, 62, 175, 340

Rename disc files, 175, 307, 340

RENUM, 13, 341

Renumbering programs, **13**, 341

RESET, 62, 174, 342

Reset disc, **62**

RESTORE, 16, 343

RESUME, 344

RETURN, 345

Return codes, 192

RIGHT\$, 56, 346

RMDIR, 175, 347

RND, 348

ROUND, 60, 349

Rounding, **60**, 134

RSET, 79, 180, 350

RUN, 351, 352

Run-only, 127, 142

Running programs, **10**, 152

S: command line parameter, 389

SAVE, 353

scaled-number, 130, 132

Scaled numbers, **21**

Screen

- displaying data on, **27**
- installation, 410

Screen control, **30**, 169

- codes, **29**

- VT52 screen, **30**

- other types of screen, **33**

Screen lines, **147**

Screen-based editor

- controls, 147

- insert, 148

- overstrike, 148

Searching strings, **58**

SEEKKEY, 99, 354

SEEKNEXT, 111, 355

SEEKPREV, 111, 356

SEEKRANK, 357

SEEKREC, 358

SEEKSET, 111, 359

Segments

- define base, 234

Segment registers, 402

Semicolon

- in print command, 164

Sequential files, **63**, 177

- changing, **68**

- closing, **68**

- creating, **63**

- example, **69**

- multiuser, 406

- opening, **63**, 313

- reading, **65**

- writing, **64**

SGN, 54, 360

Signs

- formatted printing, 168

- Single Length values
 - output, 165
- simple-variable*, 196
- SIN, **54**, 361
- Sine, 361
- Single Length values, 131
 - exact representation, 140
 - in random records, 229, 305
- Single precision numbers, **22**
- SPACE\$, **57**, 362
- SPC, **28**, 166, 332, 363
- SQR, 364
- Square root, 364
- Stack
 - initialisation, 217
- Stack size, 297
- Starting BASIC, **1**
- Statements, 127
- Step size, 261
- STOP, **44**, 365
 - resume after, 222
- Stopping a program, **44**
- Storing information
 - in the program, **16**
 - on disc, **10**
- STR\$ function, 366
- Strings, **23**, 131
 - = operator, 138
 - concatenation of, 137
 - convert to double length
 - numeric, 226
 - convert to integer, 227, 228
 - convert to lower case, **58**, 291
 - convert to numeric values, **59**, 378
 - convert to single length integer,
 - 229, 230
 - convert to upper case, **58**, 376
 - equality, 138
 - generating, **57**
 - input from keyboard, **19**
 - joining, **5**, **56**
 - left justify, 294
 - length of, **57**, 280
 - of repeated character, 367
 - of spaces, 362
 - operators, **5**
 - right justify, 350
 - searching, **58**
 - splitting, **57**
 - substrings, 276, 279, 299, 346
 - variables, **3**, **23**
 - with bit 7 stripped, 368
- string-constant*, 131
- string-expression*, 137, 197
- String functions, 159
- string-function*, 137
- string-item*, 137
- String variables, **56**
- string-variable*, 137, 197
- STRING\$, **57**, 367
- STRIP\$ function, 368
- Structure of programs, **37**
- SUB control character
 - in sequential files, 178
- Subkeys, 406
- Subroutines, **46**, 345
 - call, 265
 - calling external subroutines,
 - 208, 400
 - external subroutines, 397
 - flow, **50**
 - for console input, 318
 - for output to console, 323
 - for output to printer, 321
 - position in program, **48**
- Substrings, **56**, 276, 279, 299, 346
- Subtraction, **5**, 136
- Suspend BASIC, 142
- SWAP, 369

- Syntax error
 - entry to editor, 145
 - in run only system, 145
- SYSTEM, **1**, 370
- TAB, **28**, 166, 332, 371
 - in sequential files, 178
- Tab expansion
 - disable, 322
 - enable, 326
- TAN, **54**, 372
- Template
 - format (printing), **28**, 167
- Temporary write lock, 185
- Text strings, **23**
- THEN, **41**, 269
- Toggle insert/overstrike, 148
- Top down programming, **48**
- Tracing, 373
- Tree-structured directories, 175
- Trigonometry, **54**, 423
- TROFF, 373
- TRON, 373
- Truncation of numeric values,
 - 60**, 134, 260
- TYPE, **62**, 374
- Type conversion, 159
 - integer/floating point, **60**
 - integer/string, **59**
- Type-marker, 132, 197
- Unary minus, 136
- Unary plus, 136
- Unscaled numbers, **21**
 - unscaled-number*, 130, 132
- Unsigned integer values, 135
- UNT function, 375
- UPPER\$ function, 376
- User defined functions, **50**, 139, 233
- User segment, 234
- USING, **29**, 166, 332
- USR functions, **120**, 235, 377, 398
- VAL, **59**, 378
- variable*, 197
- variable-name*, 197
- Variables, 132
 - address of, 379
 - clear all values, 217
 - clear except common, 220
 - control variable, 261
 - declaration of, 133
 - in subroutine, **48**
 - initialising, 231, 338
 - life of, **21**
 - names, **3**, **20**, 129
 - numeric, **3**
 - set default type, 236
 - shared between programs, 211, 219
 - string, **3**, **23**, **56**
 - swap values, 369
- VARPTR, 379
- VERSION, 153, 380
- WAIT, 381
- WAITW, 381
- WEND, 382
- WHILE, 383
- White space, 129
- Widening
 - of values, 134
- WIDTH, 384, 255, 171
- WIDTH LPRINT, 385, 255, 171
- Wild cards (in file names), 174
- WRITE, 386
- Write
 - see also Output
- WRITE #, **64**, 387

XOR operator, 139

ZONE, 388

Zones, 27, 165

set size, 388

', 37

^ operator, 136

^ print format, 167, 168

! print format, 167

! type marker, 22, 132

print format, 167

type marker, 22, 132

\$ print format, 168

\$ type marker, 23, 132

& print format, 167

% type marker, 22, 132

* operator, 53, 136

* print format, 168

+ operator, 53, 136

+ print format, 168

on strings, 137

- operator, 53, 136

- print format, 168

/ operator, 53, 136

\operator, 136

\print format, 167

. print format, 167

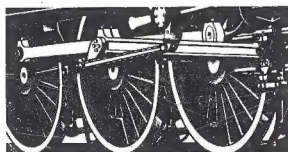
Mallard BASIC

Part 1: Introduction to Mallard BASIC

- Easy to read introduction to programming your PCW in BASIC
- How to store and manipulate data using BASIC
- Detailed chapters on sequential files, random files and “Jetsam” keyed files
- Worked examples of BASIC programs
 - including a simple database

Part 2: Reference to Mallard BASIC

- Detailed explanation of commands and concepts
- Complete A–Z of BASIC commands and functions
- Technical appendices including trigonometrical functions and details of BASIC arithmetic



**LOCOMOTIVE
SOFTWARE**

MAY 89
ISBN 1 85195 009 5



9 781851 950096